REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	10.Jun.02	DISSERTATION

4. TITLE AND SUBTITLE
CONTRIBUTIONS TO A GENERAL THEORY OF CODES

5. FUNDING NUMBERS

6. AUTHOR(S)
CAPT HOLCOMB TRAE D

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
TEXAS A&M UNIVERSITY

8. PERFORMING ORGANIZATION REPORT NUMBER

CI02-80

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

20020702 015

14. SUBJECT TERMS

15. NUMBER OF PAGES
213

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

CONTRIBUTIONS TO A GENERAL THEORY OF CODES

A Dissertation

by

TRAE DOUGLAS HOLCOMB

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2002

Major Subject: Mathematics
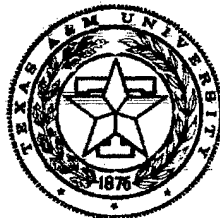
TEXAS A&M UNIVERSITY

Office of the Vice President for Research
Office of Graduate Studies
1113 TAMU • College Station, Texas 77843-1113
(979) 845-3631
FAX (979) 845-1596

23 APRIL 2002

TO WHOM IT MAY CONCERN:

This is to certify that the student named below has completed all requirements for the degree indicated.

TRAE DOUGLAS HOLCOMB
NAME

DOCTOR OF PHILOSOPHY
DEGREE

MATHEMATICS
MAJOR

10 MAY 2002
CONFERRAL DATE

John R. Giardino
Dean of Graduate Studies

CONTRIBUTIONS TO A GENERAL THEORY OF CODES

A Dissertation

by

TRAE DOUGLAS HOLCOMB

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Approved as to style and content by:

G.R. Blakley (Chair of Committee)	Itshak Borosh (Member)
Peter Stiller (Member)	Andreas Klappenecker (Member)
William Rundell (Head of Department)	

May 2002

Major Subject: Mathematics

ABSTRACT

Contributions to a General Theory of Codes. (May 2002)

Trae Douglas Holcomb, B.S., Southwest Texas State University;

M.S., University of Colorado at Colorado Springs

Chair of Advisory Committee: Dr. G. R. Blakley

In 1997, Drs. G. R. Blakley and I. Borosh published two papers whose stated purpose was to present a general formulation of the notion of a code that depends only upon a code's structure and not its functionality. In doing so, they created a further generalization–the idea of a precode. Recently, Drs. Blakley, Borosh, and A. Klappenecker have worked on interpreting the structures and results in these pioneering papers within the framework of category theory.

The purpose of this dissertation is to further the above work. In particular, we seek to accomplish the following tasks within the "general theory of codes."

(1) Rewrite the original two papers in terms of the alternate representations of precodes as bipartite digraphs and Boolean matrices.

(2) Count various types of bipartite graphs up to isomorphism, and count various classes of codes and precodes up to isomorphism.

(3) Identify many of the classical objects and morphisms from category theory within the categories of codes and precodes.

(4) Describe the various ways of constructing a code from a precode by "splitting" the precode. Identify important properties of these constructions and their interrelationship. Discuss the properties of the constructed codes with regard to the factorization of homomorphisms through them, and discuss their relationship to the code constructed from the precode by "smashing."

(5) Define a parametrization of a precode and give constructions of various parametrizations of a given precode, including a "minimal" parametrization.

iv

(6) Use the computer algebra system, Maple, to represent and display a precode and its companion, opposite, smash, split, bald-split, and various parametrizations. Implement the formulae developed for counting bipartite graphs and precodes up to isomorphism.

ACKNOWLEDGMENTS

I thank my advisor, Dr. G. R. Blakley, for allowing me to assist him in furthering the work on a general theory of codes. The topic was an optimal choice for my dissertation given its many unexplored areas and the relatively short period I had available to complete this work. I greatly appreciate the freedom granted by Dr. Blakley to attack problems that piqued my interest and the suggestions of others when my own ideas were in short supply. I also appreciate Drs. Borosh, Klappenecker, and Stiller for serving as committee members and for providing valuable insights and suggestions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE   Page

1. INTRODUCTION

The stated purpose of [2] and [3] was to present a general formulation of the notion of a code, which depends only upon a code's structure and not its functionality. The purpose of this paper is to continue this work as described in the included abstract. The body of this work is written in a manner which presupposes familiarity with the contents of [2] and [3]. However, Appendix A contains the pertinent background information from these works for the reader who does not have them on hand.

This dissertation follows the style and format of the *Bulletin (New Series) of the American Mathematical Society.*

2. PRELIMINARIES

2.1. Precodes and Codes Defined. We start with the most basic of definitions.

Definition 2.1. *Let A and B be sets. A relation r from A to B is a subset of $A \times B$. We let $DOM(r) \subseteq A$ denote the domain of r and $RAN(r) \subseteq B$ denote the range of r. A relation from A to A is said to be a relation on A. We let A^2 denote the cartesian product $A \times A$. We let $diag(A^2) = \{(a,a) \mid a \in A\}$, the diagonal of A^2. Note that $diag(A^2)$ is the identity function $i_A : A \to A$. A relation r on A is called subdiagonal if $r \subseteq diag(A^2)$. For a relation $r \subseteq A \times B$, we let $r^{\leftarrow} = \{(b,a) \in B \times A \mid (a,b) \in r\}$ denote the converse relation of r.*

Following $2B1$ and $2C1$ in [2], we have the following definition.

Definition 2.2. *Let $P_{\mathcal{A}}$ and $C_{\mathcal{A}}$ be sets, and let $e \subseteq P_{\mathcal{A}} \times C_{\mathcal{A}}$ and $d \subseteq C_{\mathcal{A}} \times P_{\mathcal{A}}$. The four-tuple $\mathcal{A} = (P_{\mathcal{A}}, C_{\mathcal{A}}, e_{\mathcal{A}}, d_{\mathcal{A}})$ is called a precode from $P_{\mathcal{A}}$ to $C_{\mathcal{A}}$. $P_{\mathcal{A}}$ and $C_{\mathcal{A}}$ are called the plaintext and codetext alphabets of $\mathcal{A}$, respectively. Furthermore, we call $e_{\mathcal{A}} \subseteq P_{\mathcal{A}} \times C_{\mathcal{A}}$ the encode relation of $\mathcal{A}$ and $d_{\mathcal{A}} \subseteq C_{\mathcal{A}} \times P_{\mathcal{A}}$ the decode relation of $\mathcal{A}$.*

The precode $\mathcal{A} = (P_{\mathcal{A}}, C_{\mathcal{A}}, e_{\mathcal{A}}, d_{\mathcal{A}})$ is said to be a code from $P_{\mathcal{A}}$ to $C_{\mathcal{A}}$ if the composite relation $d_{\mathcal{A}} \circ e_{\mathcal{A}} \subseteq P_{\mathcal{A}}^2$ is a subdiagonal relation on $P_{\mathcal{A}}$, that is, if $d_{\mathcal{A}} \circ e_{\mathcal{A}}$ is an identity partial function from $P_{\mathcal{A}}$ to $P_{\mathcal{A}}$. The relation $c_{\mathcal{A}} = e_{\mathcal{A}} \cap d_{\mathcal{A}}^{\leftarrow}$ is called the circulation relation of $\mathcal{A}$.

Notation 2.3. *If $\mathcal{A}$ is a precode, we will use $P_{\mathcal{A}}$, $C_{\mathcal{A}}$, $e_{\mathcal{A}}$, and $d_{\mathcal{A}}$ to denote its plaintext set, codetext set, encode relation, and decode relation, respectively.*

Proposition $2C3$ in [2] notes that $\mathcal{A}$ is a code if and only if

$$d_{\mathcal{A}} \circ e_{\mathcal{A}} = i_{DOM(c_{\mathcal{A}})} = diag(DOM(c_{\mathcal{A}})^2).$$

Furthermore, if $\mathcal{A}$ is a code, then $c_{\mathcal{A}}^{\leftarrow} \circ c_{\mathcal{A}} = d_{\mathcal{A}} \circ e_{\mathcal{A}}$ and $c_{\mathcal{A}}^{\leftarrow}$ is a (partial) function.

Definition 2.4. *Let $m, n \in \mathbb{Z}^+$. A precode with m plaintext elements and n codetext elements is said to be an (m,n) precode. We define an (m,n) code analogously.*

We recall that precodes can be represented by bipartite digraphs and Boolean matrices. As in Definition A.2, we use $M_{\mathcal{A}}$ to denote the synoptic codebook matrix of a precode $\mathcal{A}$.

Definition 2.5. *Let $\mathcal{A}$ be a precode with synoptic codebook matrix $M_{\mathcal{A}}$. A column (row) of $\mathcal{A}$ is said to be of type o if each of its entries is an o. A column (row) is of type e if each of its entries is either an o or an e. A column (row) is of type d if each of its entries is either an o or a d. Finally, a column (row) is of type s if it contains exactly one s entry, with each remaining entry being an o. We call a column (row) of type o an o column. We define e, d, and s columns (rows) analogously.*

Remark 2.6. *Let $\mathcal{A}$ be a precode. In terms of bipartite graphs, a column in $M_{\mathcal{A}}$ associated with $\kappa \in C$ is of type*

1) o if there are no edges incident on κ.

2) e if all of the edges incident on κ are contained in the encode relation, $e_{\mathcal{A}}$.

3) d if all of the edges incident on κ are contained in $d_{\mathcal{A}}$.

4) s if there are exactly two edges incident on κ, and these edges are $(\pi, \kappa) \in e_{\mathcal{A}}$ and $(\kappa, \pi) \in d_{\mathcal{A}}$ for some $\pi \in P_{\mathcal{A}}$.

We note that precodes may have columns which are not of any of the above four types. However, as in Theorem $5B.5$ in [1], a precode $\mathcal{A}$ is a code if and only if each of the columns in $M_{\mathcal{A}}$ is of one of these four types.

2.2. Subprecodes, Unions, and Intersections. Recall the definition of subprecodes and superprecodes from Definition A.3 and the definition of the intersection of two precodes given in Definition A.4.

Notation 2.7. *Let $\hat{\mathcal{A}}$ and $\mathcal{A}$ be precodes such that $P_{\hat{\mathcal{A}}} \subseteq P_{\mathcal{A}}$ and $C_{\hat{\mathcal{A}}} \subseteq C_{\mathcal{A}}$. If $M_{\hat{\mathcal{A}}}(\pi, \kappa) \leq M_{\mathcal{A}}(\pi, \kappa)$ for all $(\pi, \kappa) \in P_{\hat{\mathcal{A}}} \times C_{\hat{\mathcal{A}}}$, we will write $M_{\hat{\mathcal{A}}} \leq M_{\mathcal{A}}$.*

Lemma 2.8. *Let $\hat{\mathcal{A}}$ and $\mathcal{A}$ be precodes such that $P_{\hat{\mathcal{A}}} \subseteq P_{\mathcal{A}}$ and $C_{\hat{\mathcal{A}}} \subseteq C_{\mathcal{A}}$. Then $\hat{\mathcal{A}}$ is a subprecode of $\mathcal{A}$ if and only if $M_{\hat{\mathcal{A}}} \leq M_{\mathcal{A}}$.*

Proof. This is clear since in $SYBAP$, $o \le e \le s$ and $o \le d \le s$. $\qquad\square$

Definition 2.9. *The union $\mathcal{A} \cup \hat{\mathcal{A}}$ of the precodes $\mathcal{A}$ and $\hat{\mathcal{A}}$ is defined to be the precode*

$$\mathcal{A} \cup \hat{\mathcal{A}} = (P_\mathcal{A} \cup P_{\hat{\mathcal{A}}}, C_\mathcal{A} \cup C_{\hat{\mathcal{A}}}, e_\mathcal{A} \cup e_{\hat{\mathcal{A}}}, d_\mathcal{A} \cup d_{\hat{\mathcal{A}}}).$$

Notation 2.10. *Let $\mathcal{A}$ and $\hat{\mathcal{A}}$ be precodes. We define the matrices*

$$M_\mathcal{A} \wedge M_{\hat{\mathcal{A}}} : (P_\mathcal{A} \cap P_{\hat{\mathcal{A}}}) \times (C_\mathcal{A} \cap C_{\hat{\mathcal{A}}}) \longrightarrow SYBAP$$

and

$$M_\mathcal{A} \vee M_{\hat{\mathcal{A}}} : (P_\mathcal{A} \cup P_{\hat{\mathcal{A}}}) \times (C_\mathcal{A} \cup C_{\hat{\mathcal{A}}}) \longrightarrow SYBAP$$

via

$$(M_\mathcal{A} \wedge M_{\hat{\mathcal{A}}})(\pi,\kappa) = M_\mathcal{A}(\pi,\kappa) \wedge M_{\hat{\mathcal{A}}}(\pi,\kappa) \text{ for all } (\pi,\kappa) \in (P_\mathcal{A} \cap P_{\hat{\mathcal{A}}}) \times (C_\mathcal{A} \cap C_{\hat{\mathcal{A}}})$$

and

$$(M_\mathcal{A} \vee M_{\hat{\mathcal{A}}})(\pi,\kappa) = \begin{cases} M_\mathcal{A}(\pi,\kappa) \vee M_{\hat{\mathcal{A}}}(\pi,\kappa) & \text{if } (\pi,\kappa) \in (P_\mathcal{A} \cap P_{\hat{\mathcal{A}}}) \times (C_\mathcal{A} \cap C_{\hat{\mathcal{A}}}), \\ M_\mathcal{A}(\pi,\kappa) & \text{if } (\pi,\kappa) \in (P_\mathcal{A} \times C_\mathcal{A}) \backslash ((P_\mathcal{A} \cap P_{\hat{\mathcal{A}}}) \times (C_\mathcal{A} \cap C_{\hat{\mathcal{A}}})), \\ M_{\hat{\mathcal{A}}}(\pi,\kappa) & \text{if } (\pi,\kappa) \in (P_{\hat{\mathcal{A}}} \times C_{\hat{\mathcal{A}}}) \backslash ((P_\mathcal{A} \cap P_{\hat{\mathcal{A}}}) \times (C_\mathcal{A} \cap C_{\hat{\mathcal{A}}})), \\ o & \text{otherwise .} \end{cases}$$

We then have

Lemma 2.11. *If $\mathcal{A}$ and $\hat{\mathcal{A}}$ are precodes, then $M_{\mathcal{A} \cap \hat{\mathcal{A}}} = M_\mathcal{A} \wedge M_{\hat{\mathcal{A}}}$ and $M_{\mathcal{A} \cup \hat{\mathcal{A}}} = M_\mathcal{A} \vee M_{\hat{\mathcal{A}}}$.*

2.3. Companions, Self-Companion Precodes, and Nubs. Recall the definition of the companion of a precode and a self-companion precode as given in Definition A.5. It is clear that the matrix for $\mathcal{A}^{pn}$, $M_{\mathcal{A}^{pn}}$, is formed by interchanging the bits in the corresponding entries of $M_\mathcal{A}$. That is, a $d = 01$ entry becomes an $e = 10$ and vice versa. Entries of type $s = 11$ or $o = 00$ remain unchanged. In particular, we have

Lemma 2.12. *Let $\mathcal{A}$ be a precode. For $(\pi, \kappa) \in P_\mathcal{A} \times C_\mathcal{A}$,*

$$M_{\mathcal{A}^{pn}}(\pi, \kappa) = \begin{cases} s \ \ \text{if } M_\mathcal{A}(\pi, \kappa) = s, \\ e \ \ \text{if } M_\mathcal{A}(\pi, \kappa) = d, \\ d \ \ \text{if } M_\mathcal{A}(\pi, \kappa) = e, \\ o \ \ \text{if } M_\mathcal{A}(\pi, \kappa) = o. \end{cases}$$

We now give a proof of Theorem 3$\mathcal{A}$2 from [2].

Theorem 2.13. *Let $\mathcal{A}$ be a precode. Then $(\mathcal{A}^{pn})^{pn} = \mathcal{A}$, and $\mathcal{A}$ is a code if and only if $\mathcal{A}^{pn}$ is a code.*

Proof. By Lemma 2.12, $M_{(\mathcal{A}^{pn})^{pn}} = M_\mathcal{A}$. Thus, $(\mathcal{A}^{pn})^{pn} = \mathcal{A}$. Furthermore, a column in $M_\mathcal{A}$ is of type s or o if and only if the corresponding column in $M_{\mathcal{A}^{pn}}$ is of the same type. A column in $M_\mathcal{A}$ is of type e (resp. d) if and only if the corresponding column in $M_{\mathcal{A}^{pn}}$ is of type d (resp. e). Since a matrix represents a code if and only if each of its columns is of type $s, e, d,$ or o, then $\mathcal{A}$ is a code if and only if $\mathcal{A}^{pn}$ is a code. $\square$

Lemma 2.14. *A precode $\mathcal{A}$ is self-companion if and only if each entry in $M_\mathcal{A}$ is either an s or o. Hence, a code is self-companion if and only if each of its columns is of type s or o. These codes are precisely the S codes as defined in Definition 4.2.*

Proof. This is clear from Lemma 2.12 since if $\mathcal{A}$ is self-companion, then $M_{\mathcal{A}^{pn}} = M_\mathcal{A}$. $\square$

We next prove Theorem 3$\mathcal{A}$7 in [2].

Theorem 2.15. $\mathcal{A}$ *is a self-companion code if and only if* $d_{\mathcal{A}}$ *is a partial function and* $e_{\mathcal{A}} = d_{\mathcal{A}}^{\leftarrow}$.

Proof. The statement that $e_{\mathcal{A}} = d_{\mathcal{A}}^{\leftarrow}$ means precisely that $M_{\mathcal{A}}$ contains only s and o entries. That is, $e_{\mathcal{A}} = d_{\mathcal{A}}^{\leftarrow}$ if and only if $\mathcal{A}$ is a self-companion precode.

($\Rightarrow$): Now, if $\mathcal{A}$ is a self-companion code, then each column of $M_{\mathcal{A}}$ is either of type s or o. Recall that an s column contains only one s entry. Thus, for each $\kappa \in C_{\mathcal{A}}$, there is at most one element in the decode relation d. Hence, d is a partial function.

($\Leftarrow$): Suppose that $e_{\mathcal{A}} = d_{\mathcal{A}}^{\leftarrow}$ and $d_{\mathcal{A}}$ is a partial function. Since $e_{\mathcal{A}} = d_{\mathcal{A}}^{\leftarrow}$, then $M_{\mathcal{A}}$ contains only s and o entries. Since $d_{\mathcal{A}}$ is a partial function, then each s column in $M_{\mathcal{A}}$ contains precisely one s entry. Thus, $\mathcal{A}$ is a code. $\square$

Recall from Definition A.6 that the self-companion kernel of a precode $\mathcal{A}$ is the precode $N(\mathcal{A}) = (P_{\mathcal{A}}, C_{\mathcal{A}}, \kappa, \kappa^{\leftarrow})$, where $\kappa = e_{\mathcal{A}} \cap d_{\mathcal{A}}^{\leftarrow}$.

Definition 2.16. *If* $\mathcal{A}$ *is an* (m, n) *code with at least one codetext element of type* s, *then we call* $N(\mathcal{A})$ *the underlying S code of* $\mathcal{A}$, *and we denote it by* $S(\mathcal{A})$. *That is,* $S(\mathcal{A})$ *is the* (m, n) *code formed from* $\mathcal{A}$ *by keeping only the s edges in* $\mathcal{A}$.

Lemma 2.17. *For* $(\pi, \kappa) \in P_{\mathcal{A}} \times C_{\mathcal{A}}$,

$$M_{N(\mathcal{A})}(\pi, \kappa) = \begin{cases} s \text{ if } M_{\mathcal{A}}(\pi, \kappa) = s, \\ o \text{ otherwise .} \end{cases}$$

Proof. This is clear from Lemma 2.14. $\square$

We now prove Theorem 3.49 in [2].

Theorem 2.18. *Let* $\mathcal{A}$ *be a precode. The nub,* $N(\mathcal{A})$, *of* $\mathcal{A}$ *is the unique maximal self-companion subprecode of* $\mathcal{A}$. *In fact,* $N(\mathcal{A}) = \mathcal{A} \cap \mathcal{A}^{pn}$. *Consequently,* $N(\mathcal{A})^{pn} = N(\mathcal{A}^{pn})$. *Also,* $N(N(\mathcal{A})) = N(\mathcal{A})$; *that is, the nub operator is idempotent.*

Proof. We recall that a precode is self-companion if and only if each entry in its synoptic codebook matrix is either an s or o. By Lemma 2.17, it is clear that $N(\mathcal{A})$ is the unique maximal self-companion subprecode of $\mathcal{A}$. It is also clear that $M_{N(N(\mathcal{A}))} = M_{N(\mathcal{A})}$ and $M_{N(\mathcal{A})} = M_{\mathcal{A}} \wedge M_{\mathcal{A}^{pn}}$. Hence, the nub operator is idempotent and $N(\mathcal{A}) = \mathcal{A} \cap \mathcal{A}^{pn}$. $\qquad\qquad\square$

2.4. Opposites, Self-Opposite Precodes, and Hinges. Recall the definition of the opposite of a precode and a self-opposite precode as given in Definition A.7.

Since the roles of $P_{\mathcal{A}}$ and $C_{\mathcal{A}}$ are switched and the roles of $e_{\mathcal{A}}$ and $d_{\mathcal{A}}$ are also interchanged, then $M_{\mathcal{A}^{op}}$ should be related to $M_{\mathcal{A}}{}^{t}$, the transpose of $M_{\mathcal{A}}$. In particular, $M_{\mathcal{A}^{op}}$ is formed from $M_{\mathcal{A}}{}^{t}$ by interchanging the bits in each entry. That is, a $d = 01$ entry becomes an $e = 10$ and vice versa. Entries of type $s = 11$ or $o = 00$ remain unchanged. We have

Lemma 2.19. *Let $\mathcal{A}$ be a precode. For $(\pi, \kappa) \in P_{\mathcal{A}} \times C_{\mathcal{A}}$,*

$$
M_{\mathcal{A}^{op}}(\kappa, \pi) = \begin{cases}
s \text{ if } M_{\mathcal{A}}{}^{t}(\kappa, \pi) = s, \\
e \text{ if } M_{\mathcal{A}}{}^{t}(\kappa, \pi) = d, \\
d \text{ if } M_{\mathcal{A}}{}^{t}(\kappa, \pi) = e, \\
o \text{ if } M_{\mathcal{A}}{}^{t}(\kappa, \pi) = o.
\end{cases}
$$

Lemma 2.20. *Let $\mathcal{A}$ be a precode. Then $(\mathcal{A}^{op})^{op} = \mathcal{A}$ and $M_{\mathcal{A}^{op}} = (M_{\mathcal{A}^{pn}})^{t}$.*

Proof. By Lemma 2.19, $M_{(\mathcal{A}^{op})^{op}} = M_{\mathcal{A}}$, so that $(\mathcal{A}^{op})^{op} = \mathcal{A}$. By Lemma 2.12, $M_{\mathcal{A}^{op}} = (M_{\mathcal{A}^{pn}})^{t}$. $\qquad\qquad\square$

Recall from Definition A.8 that for a precode $\mathcal{A}$, the precode

$$
H = H(\mathcal{A}) = (P_{\mathcal{A}} \cap C_{\mathcal{A}}, P_{\mathcal{A}} \cap C_{\mathcal{A}}, e_{\mathcal{A}} \cap d_{\mathcal{A}}, e_{\mathcal{A}} \cap d_{\mathcal{A}}) = \mathcal{A} \cap \mathcal{A}^{op}
$$

is called the hinge of $\mathcal{A}$. Recall from Definition A.2 that we can view the entries of $M_{H(\mathcal{A})}$ as two-bit vectors and reference them accordingly. We have that $M_{H(\mathcal{A})}(\pi, \kappa)(E) = 1$ if and

only if $M_A(\pi, \kappa)(E) = 1$ and $M_A(\kappa, \pi)(D) = 1$. Similarly, $M_{H(A)}(\pi, \kappa)(D) = 1$ if and only if $M_A(\pi, \kappa)(D) = 1$ and $M_A(\kappa, \pi)(E) = 1$. Thus, we have

Lemma 2.21. *Let A be a precode. For $(\pi, \kappa) \in (P_A \cap C_A) \times (P_A \cap C_A)$,*

$$M_{H(A)}(\pi, \kappa)(E) = M_A(\pi, \kappa)(E) \wedge M_A(\kappa, \pi)(D)$$

and

$$M_{H(A)}(\pi, \kappa)(D) = M_A(\pi, \kappa)(D) \wedge M_A(\kappa, \pi)(E).$$

We now prove Theorem $3B7$ in [2].

Theorem 2.22. *If A is a precode, then $H(H(A)) = H(A)$; that is, the hinge operator is idempotent.*

Proof. The plaintext and codetext sets of $H(A)$ are equal, as are its encode and decode relations. Thus, $H(H(A)) = H(A)$. $\square$

2.5. Janiform Codes. Recall from Definition A.9 that a precode is janiform if its opposite is a code. We recall also that $M_{A^{op}}$ is formed from the transpose $M_A{}^t$ of M_A by interchanging the bits in each entry. That is, a $d = 01$ entry becomes an $e = 10$ and vice versa. Entries of type $s = 11$ or $o = 00$ remain unchanged. Notice that this means that A^{op} is a code if and only if the precode represented by $M_A{}^t$ is a code. Thus, we have

Lemma 2.23. *A precode A is janiform if and only if each of the columns of $M_A{}^t$ (i.e. each of the rows of M_A) is of type $s, e, d,$ or o.*

2.6. Self-Companion, Self-Opposite Codes. We now prove Proposition $3D1$ from [2].

Theorem 2.24. *A code $\mathcal{A}$ is self-companion and self-opposite if and only if each of the following conditions hold:*

(1) $P_\mathcal{A} = C_\mathcal{A}$

(2) $d_\mathcal{A} = e_\mathcal{A}$

(3) $e_\mathcal{A}$ is a partial involution

Proof. By Theorem 2.15, $\mathcal{A}$ is a self-companion code if and only if $d_\mathcal{A}$ is a partial function and $e_\mathcal{A} = d_\mathcal{A}^\leftarrow$. By definition, $\mathcal{A}$ is self-opposite if and only if $P_\mathcal{A} = C_\mathcal{A}$ and $d_\mathcal{A} = e_\mathcal{A}$.

($\Rightarrow$): Suppose $\mathcal{A}$ is self-companion and self-opposite. As above, $d_\mathcal{A}$ is a partial function, $e_\mathcal{A} = d_\mathcal{A}^\leftarrow$, and $e_\mathcal{A} = d_\mathcal{A}$. Thus, $e_\mathcal{A}$ is a partial function, and $e_\mathcal{A} = e_\mathcal{A}^\leftarrow$. Hence, $e_\mathcal{A}$ is a partial involution.

($\Leftarrow$): Suppose $P_\mathcal{A} = C_\mathcal{A}$, $d_\mathcal{A} = e_\mathcal{A}$, and $e_\mathcal{A}$ is a partial involution. Since $P_\mathcal{A} = C_\mathcal{A}$ and $d_\mathcal{A} = e_\mathcal{A}$, then $\mathcal{A}$ is self-opposite. Since $e_\mathcal{A}$ is a partial involution, then $e_\mathcal{A} = e_\mathcal{A}^\leftarrow = d_\mathcal{A}^\leftarrow$. Since $d_\mathcal{A} = e_\mathcal{A}$ is a partial function, then $\mathcal{A}$ is self-companion. $\square$

We also prove Proposition $3D3$ from [2].

Theorem 2.25. *Let $\mathcal{A}$ be a precode. Then $(\mathcal{A}^{op})^{pn} = (\mathcal{A}^{pn})^{op}$.*

Proof. As noted in Lemma 2.20, the matrix for the opposite of a precode is simply the transpose of the matrix of the precode's companion. Similarly, the matrix of the companion of a precode is the transpose of the matrix of the opposite. By Lemma 2.13, $(\mathcal{A}^{pn})^{pn} = \mathcal{A}$, so that $M_{(\mathcal{A}^{pn})^{op}} = (M_{(\mathcal{A}^{pn})^{pn}})^t = (M_\mathcal{A})^t$. By Lemma 2.20, $(\mathcal{A}^{op})^{op} = \mathcal{A}$, so that $M_{(\mathcal{A}^{op})^{pn}} = (M_{(\mathcal{A}^{op})^{op}})^t = (M_\mathcal{A})^t$. Thus, $(\mathcal{A}^{op})^{pn} = (\mathcal{A}^{pn})^{op}$. $\square$

Recall Definition A.10. We next prove Proposition $3D5$ from [2].

Theorem 2.26. *Let $\mathcal{A}$ be a precode. The number of members in the quartet of $\mathcal{A}$ which are codes is an even number. If $\mathcal{A}$ is a janiform code, then all four members are janiform codes.*

Proof. By Theorem A.5, $\mathcal{A}$ is a code if and only if $\mathcal{A}^{pn}$ is a code, and $\mathcal{A}^{op}$ is a code if and only if $\mathcal{A}^{oppn}$ is a code. Since a janiform code is a code whose opposite is also a code, then the theorem holds. $\square$

2.7. Nulls. Let $\mathcal{A}$ be a precode, and recall Definition A.11. In terms of matrices, we note that

1) $e_{\mathcal{A}}NL$ is the subset of $C_{\mathcal{A}}$ corresponding to the d columns of $M_{\mathcal{A}}$.

2) $d_{\mathcal{A}}NL$ is the subset of $C_{\mathcal{A}}$ corresponding to the e columns of $M_{\mathcal{A}}$.

3) $c_{\mathcal{A}}NL$ is the subset of $C_{\mathcal{A}}$ corresponding to the columns of $M_{\mathcal{A}}$ having no s entries.

4) $s_{\mathcal{A}}NL$ is the subset of $C_{\mathcal{A}}$ corresponding to the o columns of $M_{\mathcal{A}}$.

5) $e_{\mathcal{A}}VD$ is the subset of $P_{\mathcal{A}}$ corresponding to the d rows of $M_{\mathcal{A}}$.

6) $d_{\mathcal{A}}VD$ is the subset of $P_{\mathcal{A}}$ corresponding to the e rows of $M_{\mathcal{A}}$.

7) $c_{\mathcal{A}}VD$ is the subset of $P_{\mathcal{A}}$ corresponding to the rows in $M_{\mathcal{A}}$ having no s entries.

8) $s_{\mathcal{A}}VD$ is the subset of $P_{\mathcal{A}}$ corresponding to the o rows in $M_{\mathcal{A}}$.

2.8. Homomorphisms. We start with homomorphisms between relations.

2.8.1. *Relation Homomorphisms.* Recall Definitions A.12, A.13, and A.14. We now prove Proposition 7.A5 from [3].

Theorem 2.27. *Suppose that the function pair (g, h) is a relation isomorphism from (G, H, r) to $(\hat{G}, \hat{H}, m)$. Then g and h are bijections. Moreover, (g, h) and $(g^{\leftarrow}, h^{\leftarrow})$ are strong relation homomorphisms.*

Proof. Since g and $g^{\leftarrow}$ are both functions, then g is a bijection with inverse $g^{\leftarrow}$. Similarly, h is a bijection with inverse $h^{\leftarrow}$. Since $(g^{\leftarrow}, h^{\leftarrow})$ is a relation homomorphism from $(\hat{G}, \hat{H}, m)$ to (G, H, r), then $r \supseteq h^{\leftarrow} \circ m \circ g$. Thus, $h \circ r \circ g^{\leftarrow} \supseteq m$, which shows that (g, h) is a strong relation homomorphism. We similarly see that $(g^{\leftarrow}, h^{\leftarrow})$ is a strong relation homomorphism. $\square$

In the following example, we see that not every relation homomorphism (g, h) for which g and h are bijections is a relation isomorphism. However, as noted in [3], a strong relation homomorphism (g, h) for which g and h are bijections is a relation isomorphism.

Example 2.28. *Let $G = \{0, 1\}$. Note that 1_G is a bijection, but*

$$(1_G, 1_G) : (G, G, \emptyset) \longrightarrow (G, G, G \times G)$$

is a relation homomorphism which is not an isomorphism.

2.8.2. *Quotients and Canonical Maps.* Recall Definitions A.15, A.16, and A.17. We now prove Theorem 7A7 from [3].

Theorem 2.29. *Let (G, H, r) be a relation, and let (G, G, s) and (H, H, t) be equivalence relations. The canonical map pair (f_s, f_t) from the relation (G, H, r) to the relation $(G, H, r)/(s, t)$ is a strong homomorphism with kernel (s, t).*

Proof. Recall that $(G, H, r)/(s, t) = (G/s, H/t, r/(s, t))$. To show that (f_s, f_t) is a strong homomorphism, we need to show that $r/(s, t) \subseteq f_t \circ r \circ f_s^{\leftarrow}$. However, $r/(s, t) = f_t \circ r \circ f_s^{\leftarrow}$ by definition. Furthermore, the kernel of (f_s, f_t) is defined to be $(f_s^{\leftarrow} \circ f_s, f_t^{\leftarrow} \circ f_t) = (s, t)$. $\square$

2.8.3. *Isomorphism Theorems for Relations.* We can restate the second part of Theorem A.18 in a way that more closely resembles the first isomorphism theorem as given in [7].

Theorem 2.30. *Let (g, h) be a relation homomorphism from (G, H, r) to $(\hat{G}, \hat{H}, m)$ with kernel $(s, t) = (g^{\leftarrow} \circ g, h^{\leftarrow} \circ h)$. Then the natural map pair $n = (g \circ f_s^{\leftarrow}, h \circ f_t^{\leftarrow})$ is a relation isomorphism from $(G, H, r)/(s, t)$ onto $Im((g, h)) = (g(G), h(H), h \circ r \circ g^{\leftarrow})$.*

Proof. This follows directly from Theorem A.18 since (g, h) is by definition a strong relation epimorphism onto its image. $\square$

2.8.4. *Precode Homomorphisms.* Recall Definitions A.19, A.20, and A.21.

Remark 2.31. *We note that the precodes $\mathcal{A}$ and $\hat{\mathcal{A}}$ are isomorphic if and only if their bipartite digraphs are graph isomorphic, that is, if and only if the matrices M_A and $M_{\hat{A}}$ determine isomorphic graphs. Since the synoptic codebook matrix of a precode is, in essence, an incidence matrix for the precode's associated bipartite digraph, then $\mathcal{A}$ and $\hat{\mathcal{A}}$ are isomorphic if and only if there are bijections $\sigma : P_A \to P_{\hat{A}}$ and $\tau : C_A \to C_{\hat{A}}$ such that $M_A(\pi, \kappa) = M_{\hat{A}}(\sigma(\pi), \tau(\kappa))$ for all $(\pi, \kappa) \in P_A \times C_A$.*

As we saw in Example A.22, not every precode homomorphism (g, h) for which g and h are bijections is a precode isomorphism. However, we note a strong precode homomorphism (g, h) for which g and h are bijections is a precode isomorphism.

2.8.5. *Isomorphism Theorems for Precodes.* Recall Definition A.23. We now prove Theorem 9B1 from [3]. It is an analogue to the first isomorphism theorem of group theory.

Theorem 2.32. *Let $(g, h) : \mathcal{A} \longrightarrow \hat{\mathcal{A}}$ be a precode homomorphism with kernel $(s, t) = (g^{\leftarrow} \circ g, h^{\leftarrow} \circ h)$. Then the following three statements hold.*

1. *The natural map pair $n = (g \circ f_s^{\leftarrow}, h \circ f_t^{\leftarrow})$ is a precode homomorphism from $\mathcal{A}/(s, t)$ to $\hat{\mathcal{A}}$.*
2. *If (g, h) is a strong precode epimorphism, then $n : \mathcal{A}/(s, t) \longrightarrow \hat{\mathcal{A}}$ is a precode isomorphism.*
3. *If $\hat{\mathcal{A}}$ is a code, then $\mathcal{A}/(s, t)$ is also a code.*

Proof. Part 3 was shown in [3], so we need only show the other two parts. By the definition of a precode homomorphism,

$$(g, h) : (P_A, C_A, e_A) \longrightarrow (P_{\hat{A}}, C_{\hat{A}}, e_{\hat{A}}) \text{ and } (h, g) : (C_A, P_A, d_A) \longrightarrow (C_{\hat{A}}, P_{\hat{A}}, d_{\hat{A}})$$

are relation homomorphisms with kernels (s, t) and (t, s), respectively. By Theorem A.18, then $(g \circ f_s^{\leftarrow}, h \circ f_t^{\leftarrow})$ is a relation homomorphism from $(P_A, C_A, e_A)/(s, t) = (P_A/s, C_A/t, e_A/(s, t))$ to $(P_{\hat{A}}, C_{\hat{A}}, e_{\hat{A}})$ and is an isomorphism if (g, h) is a strong relation epimorphism from (P_A, C_A, e_A) to $(P_{\hat{A}}, C_{\hat{A}}, e_{\hat{A}})$. Similarly, $(h \circ f_t^{\leftarrow}, g \circ f_s^{\leftarrow})$ is a relation homomorphism from $(C_A, P_A, d_A)/(t, s) =$

$(C_A/t, P_A/s, d_A/(t,s))$ to $(C_{\hat{A}}, P_{\hat{A}}, d_{\hat{A}})$ and is an isomorphism if (h,g) is a strong relation epimorphism from (C_A, P_A, d_A) to $(C_{\hat{A}}, P_{\hat{A}}, d_{\hat{A}})$. By definition, then $(g \circ f_s^{\leftarrow}, h \circ f_t^{\leftarrow})$ is a precode homomorphism from $\mathcal{A}/(s,t)$ to $\hat{\mathcal{A}}$ which is an isomorphism if (g,h) is a strong precode epimorphism from $\mathcal{A}$ to $\hat{\mathcal{A}}$. □

We can restate the second part of this theorem in a way that more closely resembles the first isomorphism theorem as given in [7].

Theorem 2.33. *Let* $(g,h) : \mathcal{A} \longrightarrow \hat{\mathcal{A}}$ *be a precode homomorphism with kernel* $(s,t) = (g^{\leftarrow} \circ g, h^{\leftarrow} \circ h)$. *Then the natural map pair* $n = (g \circ f_s^{\leftarrow}, h \circ f_t^{\leftarrow})$ *is a precode isomorphism from* $\mathcal{A}/(s,t)$ *to* $Im((g,h)) = (g(P_A), h(C_A), h \circ e_A \circ g^{\leftarrow}, g \circ d_A \circ h^{\leftarrow})$.

Proof. This follows directly from Theorem 2.32 since (g,h) is a strong precode epimorphism onto its image. □

2.8.6. Precode Homomorphisms in Terms of Bipartite Digraphs and Matrices. We recall that precodes can be represented via bipartite digraphs, and we note that there is a natural one-to-one correspondence between precode homomorphisms and the homomorphisms of the associated bipartite digraphs. There is also a nice way of describing precode homomorphisms in terms of their effect on the matrices associated with the precodes.

Let (g,h) be a precode homomorphism from $\mathcal{A}$ to $\hat{\mathcal{A}}$ with kernel $(s,t) = (g^{\leftarrow} \circ g, h^{\leftarrow} \circ h)$. By Theorem 2.33,

$$(g \circ f_s^{\leftarrow}, h \circ f_t^{\leftarrow}) : \mathcal{A}/(s,t) \longrightarrow Im((g,h)) = (g(P), h(C), h \circ e \circ g^{\leftarrow}, g \circ d \circ h^{\leftarrow})$$

is a precode isomorphism. So, a matrix representation of the precode $\mathcal{A}/(s,t)$ is also a matrix representation for $Im((g,h))$.

We further note that for any equivalence relations (P_A, P_A, s) and (C_A, C_A, t), we have the strong precode epimorphism $(f_s, f_t) : \mathcal{A} \to \mathcal{A}/(s,t)$, where $f_s : P_A \to P_A/s$ and $f_t : C_A \to C_A/t$ are the canonical maps. Hence, $\mathcal{A}/(s,t) = Im((f_s, f_t))$.

Thus, the matrix representations of homomorphic images of a precode are precisely those for the precodes of the form $\mathcal{A}/(s,t)$, where s and t are equivalence relations on $P_\mathcal{A}$ and $C_\mathcal{A}$, respectively. Recall that $\mathcal{A}/(s,t) = (P_\mathcal{A}/s, C_\mathcal{A}/t, e_\mathcal{A}/(s,t), d_\mathcal{A}/(t,s))$. Recall also that

$$
\begin{aligned}
e_\mathcal{A}/(s,t) &= f_t \circ e_\mathcal{A} \circ f_s^{\leftarrow} \\
&= \{(s(\{\gamma\}), t(\{\eta\})) \mid (\gamma, \eta) \in e_\mathcal{A}\};
\end{aligned}
$$

that is, for $\alpha \in P_\mathcal{A}/s$ and $\beta \in C_\mathcal{A}/t$,

$$
(\alpha, \beta) \in e_\mathcal{A}/(s,t) \Leftrightarrow \text{ there are } \pi \in \alpha \text{ and } \kappa \in \beta \text{ such that } (\pi, \kappa) \in e_\mathcal{A}.
$$

In matrix terms,

$$
M_{\mathcal{A}/(s,t)}(\alpha, \beta)(E) = 1 \Leftrightarrow M_\mathcal{A}(\pi, \kappa)(E) = 1 \text{ for some } \pi \in \alpha \text{ and } \kappa \in \beta.
$$

Thus, $M_{\mathcal{A}/(s,t)}(\alpha, \beta)(E) = \bigvee_{\pi \in \alpha} \bigvee_{\kappa \in \beta} M_\mathcal{A}(\pi, \kappa)(E)$, where the $\vee$ is the $ZOBAP$ operator. Similarly, $M_{\mathcal{A}/(s,t)}(\alpha, \beta)(D) = \bigvee_{\pi \in \alpha} \bigvee_{\kappa \in \beta} M_\mathcal{A}(\pi, \kappa)(D)$. This gives us the following theorem.

Theorem 2.34. *Let $\mathcal{A}$ be a precode, and let $(P_\mathcal{A}, P_\mathcal{A}, s)$ and $(C_\mathcal{A}, C_\mathcal{A}, t)$ be equivalence relations. Let $M_\mathcal{A}$ be a matrix representation of $\mathcal{A}$. Then a matrix representation for $\mathcal{A}/(s,t)$ can be defined via $M_{\mathcal{A}/(s,t)}(\alpha, \beta) = \bigvee_{\pi \in \alpha} \bigvee_{\kappa \in \beta} M_\mathcal{A}(\pi, \kappa)$, where $\vee$ is the $SYBAP$ operator.*

Proof. We merely recall that we can view each of o, d, e, and s as two-vectors over $ZOBAP$, in which case the $SYBAP$ operation $\vee$ is precisely the corresponding operation in $ZOBAP$ applied coordinate-wise. $\qquad\square$

It is clear that there is a homomorphism from $\mathcal{A}$ to $\hat{\mathcal{A}}$ if and only if there is a subprecode $\mathcal{B} = (P_\mathcal{B}, C_\mathcal{B}, e_\mathcal{B}, d_\mathcal{B})$ of $\hat{\mathcal{A}}$ which is isomorphic to $\mathcal{A}/(s,t)$ for some equivalence relations $(P_\mathcal{A}, P_\mathcal{A}, s)$ and $(C_\mathcal{A}, C_\mathcal{A}, t)$. Recall that $\mathcal{B}$ and $\mathcal{A}/(s,t)$ are isomorphic if and only if there are bijections $\sigma : P_\mathcal{B} \to P_\mathcal{A}/s$ and $\tau : C_\mathcal{B} \to C_\mathcal{A}/t$ such that $M_\mathcal{B}(\pi, \kappa) = M_{\mathcal{A}/(s,t)}(\sigma(\pi), \tau(\kappa))$ for all $(\pi, \kappa) \in P_\mathcal{B} \times C_\mathcal{B}$. This proves the following theorem.

Theorem 2.35. *Let $\mathcal{A}$ and $\hat{\mathcal{A}}$ be precodes. There is a precode homomorphism from $\mathcal{A}$ to $\hat{\mathcal{A}}$ if and only if there exist a subprecode $\mathcal{B} = (P_{\mathcal{B}}, C_{\mathcal{B}}, e_{\mathcal{B}}, d_{\mathcal{B}})$ of $\hat{\mathcal{A}}$, equivalence relations $(P_{\mathcal{A}}, P_{\mathcal{A}}, s)$ and $(C_{\mathcal{A}}, C_{\mathcal{A}}, t)$, and bijections $\sigma : P_{\mathcal{B}} \to P_{\mathcal{A}}/s$ and $\tau : C_{\mathcal{B}} \to C_{\mathcal{A}}/t$ such that $M_{\mathcal{B}}(\pi, \kappa) = M_{\mathcal{A}/(s,t)}(\sigma(\pi), \tau(\kappa))$ for all $(\pi, \kappa) \in P_{\mathcal{B}} \times C_{\mathcal{B}}$.*

2.9. Products and Sums. Let $\{\mathcal{A}_i = (P_i, C_i, e_i, d_i)\}_{i \in I}$ be a family of precodes indexed by a set I. Recall that as in Definition A.24, the product of the $\mathcal{A}_i$ is the precode

$$\mathcal{A} = (P_{\mathcal{A}} = \prod_{i \in I} P_i, C_{\mathcal{A}} = \prod_{i \in I} C_i, e_{\mathcal{A}} = \prod_{i \in I} e_i, d_{\mathcal{A}} = \prod_{i \in I} d_i).$$

For $\pi = \prod_{i \in I} \pi_i \in P_{\mathcal{A}}$ and $\kappa = \prod_{i \in I} \kappa_i \in C_{\mathcal{A}}$, we have that

$$(\pi, \kappa) \in e_{\mathcal{A}} \Leftrightarrow (\pi_i, \kappa_i) \in e_i \text{ for each } i \in I,$$

which implies that

$$M_{\mathcal{A}}(\pi, \kappa)(E) = 1 \Leftrightarrow M_{\mathcal{A}_i}(\pi, \kappa)(E) = 1 \text{ for each } i \in I.$$

Thus, $M_{\mathcal{A}}(\pi, \kappa)(E) = \bigwedge_{i \in I} M_{\mathcal{A}_i}(\pi_i, \kappa_i)(E)$. Similarly, $M_{\mathcal{A}}(\pi, \kappa)(D) = \bigwedge_{i \in I} M_{\mathcal{A}_i}(\pi_i, \kappa_i)(D)$. Since the $SYBAP$ operation $\wedge$ is precisely the corresponding operation in $ZOBAP$ applied coordinate-wise, then we have

Lemma 2.36. *Let $\{\mathcal{A}_i = (P_i, C_i, e_i, d_i)\}_{i \in I}$ be a family of precodes indexed by a set I, with product*

$$\mathcal{A} = (P_{\mathcal{A}} = \prod_{i \in I} P_i, C_{\mathcal{A}} = \prod_{i \in I} C_i, e_{\mathcal{A}} = \prod_{i \in I} e_i, d_{\mathcal{A}} = \prod_{i \in I} d_i).$$

For $\pi = \prod_{i \in I} \pi_i \in P_{\mathcal{A}}$ and $\kappa = \prod_{i \in I} \kappa_i \in C_{\mathcal{A}}$, $M_{\mathcal{A}}(\pi, \kappa) = \bigwedge_{i \in I} M_{\mathcal{A}_i}(\pi_i, \kappa_i)$.

Recall Definition A.25. We clearly have

Lemma 2.37. *Let $\{\mathcal{A}_i = (P_i, C_i, e_i, d_i)\}_{i \in I}$ be a family of precodes indexed by a set I, with direct sum*

$$\mathcal{A} = (P_{\mathcal{A}} = \bigcup_{i \in I} P_i \times \{i\}, C_{\mathcal{A}} = \bigcup_{i \in I} C_i \times \{i\}, e_{\mathcal{A}} = \bigcup_{i \in I} e_i \times \Delta_i, d_{\mathcal{A}} = \bigcup_{i \in I} d_i \times \Delta_i),$$

where Δ_i denotes the relation $\Delta_i = \{(i,i)\}$. For $((\pi,i),(\kappa,j)) \in P_{\mathcal{A}} \times C_{\mathcal{A}}$,

$$M_{\mathcal{A}}((\pi,i),(\kappa,j)) = \begin{cases} o \text{ if } i \neq j, \\ \\ M_{\mathcal{A}_i}(\pi,\kappa) \text{ if } i = j. \end{cases}$$

3. ENUMERATING BIPARTITE GRAPHS

In Sections 4 and 5, we show how to count codes and precodes up to isomorphism. To do so, we need to count certain types of bipartite graphs up to equivalence. In particular, we need to count bipartite graphs with m vertices of one color and n vertices of another up to equivalence, where equivalence is as defined in Definition 3.10. We must also count mixed bipartite graphs up to isomorphism. We start with some background from [5].

Definition 3.1. *A bipartite graph is a graph that can be bicolored; that is, its vertex set can be partitioned into two disjoint, nonempty subsets such that no two adjacent vertices are contained in the same subset.*

Let $m, n \in \mathbb{Z}^+$. A bipartite graph with m vertices of one color and n of the other color is said to be an (m, n) bipartite graph.

Definition 3.2. *Let A be a permutation group with object set $X = \{1, 2, ..., n\}$. Recall that each $\alpha \in A$ can be written uniquely as a product of disjoint cycles. For each $1 \leq k \leq n$, we let $j_k(\alpha)$ denote the number of cycles of length k in the cycle decomposition of α. The cycle index of A is the polynomial in the variables $s_1, ..., s_n$ defined by*

$$Z(A) = Z(A; s_1, ..., s_n) = \frac{1}{|A|} \sum_{\alpha \in A} \prod_{1 \leq k \leq n} s_k^{j_k(\alpha)}.$$

Furthermore, $Z(A, 1 + x)$ is defined to be $Z(A; 1 + x, 1 + x^2, ...)$, where we substitute $1 + x^k$ for s_k in the formula for $Z(A)$.

The following is Corollary 2.5.1 of the Pólya Enumeration Theorem in [5].

Theorem 3.3. *Let A be a permutation group with object set X. The coefficient of x^r in $Z(A, 1+x)$ is the number of A equivalence classes of r sets of X.*

We now establish some notation that we will employ throughout the remainder of this section.

Notation 3.4. *As is commonly done, we use S_k to denote the symmetric group on k symbols, and we use $[r, t]$ and (r, t) to denote the least common multiple and greatest common divisor, respectively, of r and t. For $m, n \in \mathbb{Z}^+$, we let $K_{m,n}$ denote the complete (m, n) bipartite graph, and we let X be the edge set of $K_{m,n}$.*

Note that any (m, n) bipartite graph can be viewed as a spanning subgraph of $K_{m,n}$. Recall that a permutation $\phi = (\alpha, \beta) \in S_m \times S_n$ is an isomorphism between two (m, n) bipartite graphs G_1 and G_2 if and only if ϕ preserves edge adjacency. Furthermore, each $\phi = (\alpha, \beta) \in S_m \times S_n$ induces a permutation ϕ' on X via $\phi'(\{i, j\}) = \{\alpha(i), \beta(j)\}$ for every $\{i, j\} \in X$. Although we are slightly abusing notation, we will also write $\phi' = (\alpha, \beta)$.

Definition 3.5. *The collection of permutations on X induced (as described in Notation 3.4) by the permutations in $S_m \times S_n$ is a group, called the edge group of $K_{m,n}$.*

Let $\mathcal{A}$ be a subgroup of $S_m \times S_n$, and let A be the subgroup of the edge group of $K_{m,n}$ induced by the permutations in $\mathcal{A}$. By Theorem 3.3, the coefficient of x^r in $Z(A, 1 + x)$ is the number of A equivalence classes of r sets of X. Each such equivalence class (of sets of r edges in X) corresponds precisely to the set of spanning subgraphs of $K_{m,n}$ with precisely r edges which are isomorphic to each other via an element of A. We say that two such graphs are A-equivalent.

The following is an adaptation of the theorem given on page 84 of [5] for bipartite graphs and arbitrary subgroups of the edge group of $K_{m,n}$.

Theorem 3.6. *Let $\mathcal{A}$ be a subgroup of $S_m \times S_n$, and let A be the subgroup of the edge group of $K_{m,n}$ induced by the permutations in $\mathcal{A}$. The polynomial $b_{m,n,A}(x)$ which enumerates (m, n) bipartite graphs up to A-equivalence by number of edges is given by*

$$b_{m,n,A}(x) = Z(A, 1 + x),$$

where

$$Z(A) = \frac{1}{|A|} \sum_{(\alpha,\beta)\in A} \prod_{1\le r\le m; 1\le t\le n} s_{[r,t]}^{(r,t)j_r(\alpha)j_t(\beta)}.$$

Proof. We need only show that the formula for $Z(A)$ is correct. Suppose that the vertex set of $K_{m,n}$ is $V = V_1 \cup V_2$, where $V_1 = \{a_1, ..., a_m\}$ is the set of vertices of one color and $V_2 = \{b_1, ..., b_n\}$ is the set of vertices of the other color. The edge set of $K_{m,n}$ is $X = \{\{a,b\}|a \in V_1 \text{ and } b \in V_2\}$.

We note that A contains the permutations on X induced by the automorphisms of $K_{m,n}$ of the form $\phi = (\alpha,\beta) \in \mathcal{A}$, where $\alpha \in S_m$ and $\beta \in S_n$. Recall that we also use (α,β) to denote the induced permutation ϕ' on X.

Notice that $|X| = m \cdot n$. By Definition 3.2,

$$Z(A) = Z(A; s_1, ..., s_{m\cdot n}) = \frac{1}{|A|} \sum_{(\alpha,\beta)\in A} \prod_{1\le k\le m\cdot n} s_k^{j_k((\alpha,\beta))},$$

where for each $1 \le k \le m \cdot n$ and for each $(\alpha,\beta) \in A$, $j_k((\alpha,\beta))$ is the number of cycles of length k in the cycle decomposition of (α,β).

The cycle structure of any $(\alpha,\beta) \in A$ is completely determined by the cycle structures of α and β. That is, the correspondence between the permutations in $\mathcal{A}$ and A induces a correspondence between the contribution of $\phi = (\alpha,\beta)$ to $Z(\mathcal{A})$ and the contribution of $\phi' = (\alpha,\beta)$ to $Z(A)$.

Let $\phi = (\alpha,\beta) \in \mathcal{A}$, whose contribution to $Z(\mathcal{A})$ is $\prod_{1\le k\le (m\cdot n)} s_k^{j_k(\phi)}$. Ostensibly, there are two contributions made by ϕ' to the corresponding term in $Z(A)$. The first comes from edges whose endpoints are both contained in a single cycle of ϕ. The second comes from edges whose ends are contained in different cycles of ϕ. However, if $\{a,b\} \in X$ (where $a \in V_1$ and $b \in V_2$), then a and b must be in different cycles of ϕ since the elements of $\mathcal{A}$ permute the elements of V_1 and permute the elements of V_2. Thus, the only contribution is of the second form.

To calculate this contribution, let $\{a,b\} \in X$, where $a \in V_1$, $b \in V_2$, and such that a and b are in different cycles of ϕ. Let z_r be the cycle of length r containing a, and let z_t be the cycle of length t containing b. Then z_r and z_t induce a cycle of length $[r,t]$ on the edge $\{a,b\}$. There are $r \cdot t$ edges

which are affected by the two cycles. Thus, there are $((r \cdot t)/[r,t]) = (r,t)$ cycles induced by z_r and z_t.

Thus, our formula becomes

$$
\begin{aligned}
Z(A; s_1, ..., s_{m \cdot n}) &= \frac{1}{|A|} \sum_{(\alpha,\beta) \in A} \prod_{1 \le k \le m \cdot n} s_k^{j_k((\alpha,\beta))} \\
&= \frac{1}{|A|} \sum_{(\alpha,\beta) \in A} \prod_{1 \le r \le m; 1 \le t \le n} s_{[r,t]}^{(r,t)j_r(\alpha)j_t(\beta)},
\end{aligned}
$$

where $j_r(\alpha)$ is the number of cycles of length r in the cycle decomposition of α and $j_t(\beta)$ is the number of cycles of length t in the cycle decomposition of β. $\square$

We note (as in the comments on page 99 of [5]) that the above theorem holds when $m = n$ only if the colors are not interchangeable. However, this is precisely the case in which we are interested, since the bipartite graphs associated with codes are digraphs. We give the following example as an illustration.

Example 3.7. *Let* $m = n = 2$. *Consider* $\mathcal{A} = (P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A}, d_\mathcal{A})$ *and* $\mathcal{B} = (P_\mathcal{B}, C_\mathcal{B}, e_\mathcal{B}, d_\mathcal{B})$, *where* $P_\mathcal{A} = \{p_1, p_2\} = P_\mathcal{B}$, $C_\mathcal{A} = \{c_1, c_2\} = C_\mathcal{B}$, $e_\mathcal{A} = \{(p_1, c_1), (p_1, c_2)\}$, $e_\mathcal{B} = \{(p_1, c_1), (p_2, c_1)\}$, *and* $d_\mathcal{A} = \emptyset = d_\mathcal{B}$, *as depicted in Figure 1.*
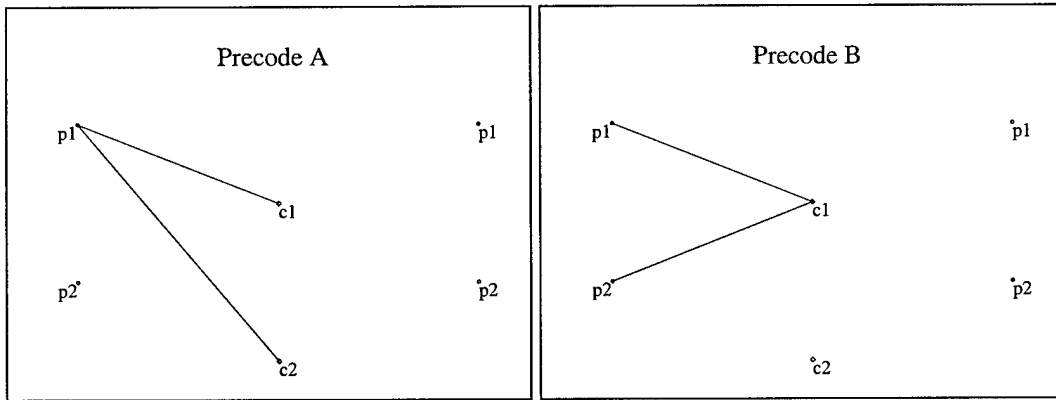
FIGURE 1. Nonisomorphic Codes with Isomorphic Bipartite Graphs

Although the undirected bipartite graphs for these codes are isomorphic, the codes are not.

3.1. Counting Bipartite Graphs up to M-equivalence. We begin with a definition.

Definition 3.8. *Let $n \in \mathbb{Z}^+$. A nonnegative partition of n of length k is a k-tuple $(n_1, ..., n_k)$ of nonnegative integers where repetition is allowed; the order of the integers does not matter; and $n = \sum_{1 \leq j \leq k} n_j$. A partition of n of length k is a nonnegative partition of n of length k, where each $n_j \in \mathbb{Z}^+$. A nonnegative composition of n of length k is an ordered k-tuple $(n_1, ... n_k)$ of nonnegative integers where repetition is allowed and $n = \sum_{1 \leq j \leq k} n_j$. A composition of n of length k is a nonnegative composition of n of length k, where each $n_j \in \mathbb{Z}^+$. We let $part(n)$ and $npart(n)$ denote the sets of partitions and nonnegative partitions, respectively, of n. Similarly, we let $comp(n)$ and $ncomp(n)$ denote the sets of compositions and nonnegative compositions, respectively, of n.*

Definition 3.9. *Let $M = (p_1, ..., p_k)$ be a given partition of $m \in \mathbb{Z}^+$. Then $\alpha \in S_m$ is said to be a permutation of m of type M if α can be written as $\alpha = \alpha_1 \cdots \alpha_k$, where $\alpha_j \in S_{p_j}$ for each $1 \leq j \leq k$. We will consider α to be an element of $S_{p_1} \times \cdots \times S_{p_k}$, and we may write $\alpha = (\alpha_1, \ldots, \alpha_k)$.*

Definition 3.10. *Let M be a given partition of $m \in \mathbb{Z}^+$. Then two bipartite (m, n) graphs G_1 and G_2 are said to be equivalent with respect to M (or M-equivalent) if there is a graph isomorphism between G_1 and G_2 of the form $\phi = (\alpha, \beta) \in S_m \times S_n$, where $\alpha \in S_m$ is a permutation of type M and $\beta \in S_n$.*

Notation 3.11. *In the discussion that follows, we fix a partition $M = (p_1, \ldots, p_k)$ of $m \in \mathbb{Z}^+$, and we let A denote the set of permutations on X induced by the automorphisms of $K_{m,n}$ which are of type M. It is clear that A is a subgroup of the edge group of $K_{m,n}$.*

We let $b_{m,n,M}(x)$ denote the polynomial which enumerates the M-equivalence classes of (m, n) bipartite graphs. That is, the coefficient of x^r in $b_{m,n,M}(x)$ is the number of M-equivalence classes of (m, n) bipartite graphs which have exactly r edges.

Corollary 3.12. $b_{m,n,M}(x) = Z(A, 1 + x)$, *where*

$$Z(A) = \frac{1}{p_1! p_2! \cdots p_k! n!} \sum_{(\alpha,\beta) \in A} \prod_{1 \leq r \leq m; 1 \leq t \leq n} s_{[r,t]}^{(r,t)j_r(\alpha)j_t(\beta)}.$$

Proof. This follows immediately from Theorem 3.6 since $|A| = p_1! \cdots p_k! n!$. □

Remark 3.13. *We will frequently use* $b_{m,n,M}$ *to denote* $b_{m,n,M}(1)$, *the number of* (m,n) *bipartite graphs up to M-equivalence. Furthermore, if* $M = (m)$, *we will often use* $b_{m,n}$ *instead of* $b_{m,n,M}$. *To compute* $b_{m,n,M}$, *we need only evaluate* $Z(A, 1 + x)$ *at* $x = 1$. *That is, we need only make the substitution* $s_k = 2$ *in* $Z(A)$ *for each* k.

Notice that if $M = (m)$, we have the following formula (given on page 96 of [5]) for counting bipartite graphs up to isomorphism.

Corollary 3.14. $b_{m,n}(x) = Z(S_m \times S_n, 1 + x)$, *where*

$$Z(S_m \times S_n) = \frac{1}{m! n!} \sum_{(\alpha,\beta) \in S_m \times S_n} \prod_{1 \leq r \leq m; 1 \leq t \leq n} s_{[r,t]}^{(r,t)j_r(\alpha)j_t(\beta)}.$$

Proof. This follows immediately from Theorem 3.6 since $|S_m \times S_n| = m! \cdot n!$. □

Example 3.15. *In* [5], *Harary computes* $b_{3,2,(3)}(x)$. *To illustrate Corollary 3.14 when* $m = n = 3$, *we compute* $b_{3,3,(3)}(x)$. *We first need to find the cycle index of the edge group of* $K_{m,n}$, $Z(S_3 \times S_3)$. *Now, there are thirty-six elements in* $S_3 \times S_3$, *but the cycle structure of any* $(\alpha, \beta) \in S_3 \times S_3$ *is completely determined by the cycle structures of* α *and* β.

For each permutation $\alpha \in S_n$ *and for each* $1 \leq k \leq n$, *recall that* $j_k(\alpha)$ *is the number of cycles of length* k *in the disjoint cycle decomposition of* α. *We can therefore naturally associate* α *with the partition of* n *which has exactly* $j_k(\alpha)$ *parts equal to* k, *and we represent this partition by the tuple* $(j(\alpha)) = (j_1(\alpha), j_2(\alpha), ..., j_n(\alpha))$. *For each partition* (j) *of* n, *we let* $h(j)$ *denote the number*

of elements in S_n with cycle decomposition corresponding to (j). We note that if $\sigma, \tau \in S_n$ have the same associated partition (j), then $(j(\sigma)) = (j(\tau))$.

Definition 3.2 gives us the following formula for $Z(S_n)$ in terms of partitions of n:

$$Z(S_n; s_1, ..., s_n) = \frac{1}{|S_n|} \sum_{\alpha \in A} \prod_{1 \leq k \leq n} s_k^{j_k(\alpha)}$$

$$= \frac{1}{n!} \sum_{(j)} h(j) \prod_{1 \leq k \leq n} s_k^{j_k},$$

where the last sum is taken over all partitions of n, and where j_k is the common value $j_k(\alpha)$, where α is any permutation with associated partition (j).

Now, if $\alpha \in S_3$ contributes the monomial $s_1 s_2$ to $Z(S_3)$ and $\beta \in S_3$ contributes s_3 to $Z(S_3)$, then $j_1(\alpha) = 1$, $j_2(\alpha) = 1$, $j_3(\alpha) = 0$, $j_1(\beta) = 0$, $j_2(\beta) = 0$, and $j_3(\beta) = 1$. Using the formula for the cycle index given in Definition 3.2, the contribution of (α, β) to $Z(S_3 \times S_3)$ is

$$\frac{1}{3!3!} \prod_{1 \leq r \leq 3; 1 \leq t \leq 3} s_{[r,t]}^{(r,t)j_r(\alpha)j_t(\beta)} = s_3 s_6.$$

Since there are three permutations with the same cycle structure as α and two permutations with the same cycle structure as β, then there are six permutation pairs of the form (σ, τ), where σ has the same cycle structure as α and τ has the same structure as β. There are also six pairs of the form (τ, σ). The total contribution of these twelve pairs to $Z(S_3 \times S_3)$ is $12 s_3 s_6$.

There are only three partitions of 3, so we have only $3 \cdot 3 = 9$ types of permutation pairs to consider. After computing the other seven, we find that

$$Z(S_3 \times S_3) = \frac{1}{3!3!} \sum_{(\alpha,\beta) \in S_m \times S_n} \prod_{1 \leq r \leq 3; 1 \leq t \leq 3} s_{[r,t]}^{(r,t)j_r(\alpha)j_t(\beta)}$$

$$= \frac{1}{36}(s_1^9 + 6 s_1^3 s_2^3 + 8 s_3^3 + 9 s_1 s_2^4 + 12 s_3 s_6).$$

To find $b_{3,3,(3)}(x) = Z(S_3 \times S_3, 1 + x)$, we must substitute $1 + x^k$ for s_k in the above expression. However, since we are only interested in the value $b_{3,3,(3)}(1)$, we need only make the substitution

$s_k = 2$ *for each k. Doing so, we find that* $b_{3,3,(3)}(1) = 36$. *This is precisely the number of* $(3,3)$ *bipartite graphs up to isomorphism, where the colors are not allowed to be interchanged.*

Recall from Remark 3.13 that to compute $b_{m,n,M}$, we need only evaluate $Z(A, 1+x)$ (as given in Corollary 3.12) at $x = 1$. That is, we need only make the substitution $s_k = 2$ in $Z(A)$ for each $1 \le k \le m \cdot n$. We now develop a formula for $b_{m,n,M}$ more suitable for implementation by a computer.

Notation 3.16. *Recall that* $M = (p_1, \dots, p_k)$ *is a fixed partition of* $m \in \mathbb{Z}^+$, *and A is the set of permutations on X induced by the automorphisms of* $K_{m,n}$ *which are of type M. For each* $1 \le i \le k$, *we let* $J[i]$ *be a list of the partitions of* p_i. *Each such partition represents one of the possible cycle structures for the elements of* S_{p_i}. *We let* $J_Z[i]$ *be the list of length* $|J[i]|$ *such that for each* $1 \le j \le |J[i]|$, $J_Z[i][j]$ *contains the list of length m which represents the cycle structure shared by the permutations of type* $J[i][j]$; *that is, for each* $1 \le q \le m$, $J_Z[i][j][q]$ *contains the number of times q appears in the partition* $J[i][j]$. *For example, if* $m = 8$, $p_i = 7$, *and* $J[i][j] = [1,1,2,3]$, *then* $J_Z[i][j] = [2,1,1,0,0,0,0,0]$. *Finally, we let* $J_N[i]$ *be the list of length* $|J[i]|$ *such that* $J_N[i][j]$ *is the number of permutations in* S_{p_i} *with cycle structure* $J_Z[i][j]$.

There are $Q = |J[1]| \cdot |J[2]| \cdots |J[k]|$ *possible cycle structures for the elements in* $S_{p_1} \times \dots \times S_{p_k}$. *We let* L_Z *and* L_N *be lists of length Q. For each* $1 \le j \le Q$, $L_Z[j]$ *contains a list of length m which represents one of the possible cycle structures for the elements in* $S_{p_1} \times \dots \times S_{p_k}$. *That is, for each* $1 \le q \le m$, $L_Z[j][q]$ *contains the number of cycles of length q in the corresponding permutation. Each entry in* L_N *contains the number of permutations of the type represented by the corresponding entry in* L_Z.

We let K be a list of the partitions of n. Each such partition represents one of the possible cycle structures for the elements of S_n. We let K_Z be the list of length $|K|$ such that for each $1 \le j \le |K|$, $K_Z[j]$ contains the list of length n which represents the cycle structure shared by the permutations of type $K[j]$; that is, for each $1 \le q \le n$, $K_Z[j][q]$ contains the number of times q

appears in the partition $K[j]$. Finally, we let K_N be the list of length $|K|$ such that $K_N[j]$ is the number of permutations in S_n with cycle structure $K_Z[j]$.

Corollary 3.17.

$$b_{m,n,M} = \frac{1}{p_1! p_2! \cdots p_k! n!} \sum_{1 \le a \le Q} \sum_{1 \le b \le |K|} \left(L_N[a] \cdot K_N[b] \cdot \left(\prod_{1 \le r \le m} \prod_{1 \le t \le n} 2^{(r,t) \cdot L_Z[a][r] \cdot K_Z[b][t]} \right) \right).$$

Proof. This follows immediately from Corollary 3.12 and Notation 3.16.

Definition 3.18. *Let $m, n \in \mathbb{Z}^+$. An (m,n) bipartite graph is said to be a 2-strict (m,n) bipartite graph if none of the n vertices of the second color are isolated. That is, there must be at least one edge incident on each of those n vertices. We define a 1-strict (m,n) bipartite graph analogously.*

Let M be a partition of m. We let $bStr_{m,n,M}$ denote the number of 2-strict (m,n) bipartite graphs up to M-equivalence, where the equivalence is as defined in Definition 3.10. If $M = (m)$, we will often use $bStr_{m,n}$ instead of $bStr_{m,n,M}$.

Let $B_{m,n}$ denote the number of (m,n) bipartite graphs with no isolated vertices (that is, (m,n) bipartite graphs which are both 1-strict and 2-strict) up to isomorphism.

Corollary 3.19. *Let M be a partition of $m \in \mathbb{Z}^+$. Then $bStr_{m,n,M}$ is given recursively by*

$$bStr_{m,n,M} = b_{m,n,M} - 1 - \sum_{1 \le k \le n-1} bStr_{m,(n-k),M}.$$

Proof. Any (m,n) bipartite graph which is not 2-strict has k isolated vertices of the second color for some $1 \le k \le n$. If $k = n$, the graph is the unique (m,n) bipartite graph with no edges. If $k \le n-1$, then the graph formed by removing the k isolated vertices is a 2-strict $(m, n-k)$ bipartite graph. Hence, the number of (m,n) bipartite graphs up to M-equivalence which are not 2-strict is $1 + \sum_{1 \le k \le n-1} bStr_{m,(n-k),M}$. $\square$

Corollary 3.20. *For $m, n \in \mathbb{Z}^+$, $B_{m,n}$ is given recursively by*

$$B_{m,n} = b_{m,n} - \sum_{1 \le j \le (m-1)} \sum_{1 \le k \le (n-1)} B_{j,k} - \sum_{1 \le j \le (m-1)} B_{j,n} - \sum_{1 \le k \le (n-1)} B_{m,k}.$$

Proof. The number of (m, n) bipartite graphs with no isolated vertices up to isomorphism is the number of (m, n) bipartite graphs up to isomorphism minus the number of (m, n) bipartite graphs with isolated vertices up to isomorphism. The number of graphs with at least one of the m vertices isolated and with none of the n vertices isolated is $\sum_{1 \leq j \leq m-1} B_{j,n}$. The number of graphs with at least one of the n vertices isolated and with none of the m vertices isolated is $\sum_{1 \leq k \leq n-1} B_{m,k}$. Finally, the number of graphs with at least one of the m and at least one of the n vertices isolated is $\sum_{1 \leq j \leq m-1} \sum_{1 \leq k \leq n-1} B_{j,k}$. $\qquad\square$

3.2. **Counting Mixed Bipartite Graphs up to Isomorphism.** To count precodes, we will also need to count mixed bipartite graphs up to isomorphism. We begin with a definition.

Definition 3.21. *A mixed graph is a graph that may contain both directed and non-directed edges.*

Notation 3.22. *We now let X denote the set of all ordered pairs corresponding to the edges of $K_{m,n}$. That is, if $\{i, j\}$ is an edge in $K_{m,n}$, then $(i, j), (j, i) \in X$.*

Each $\phi = (\alpha, \beta) \in S_m \times S_n$ induces a permutation ϕ' on X via $\phi'((i, j)) = (\alpha(i), \beta(j))$ for every $(i, j) \in X$. Although slightly abusing notation, we will also write $\phi' = (\alpha, \beta)$.

Let A denote the set of permutations on X induced by the automorphisms of $K_{m,n}$. A is called the reduced ordered pair group of $S_m \times S_n$.

Theorem 3.23. *The counting polynomial for mixed (m, n) bipartite graphs is given by*

$$m_{m,n}(x, y) = Z(A, (1 + 2x + y)^{1/2}),$$

where

$$Z(A) = \frac{1}{m!n!} \sum_{(\alpha, \beta) \in A} \prod_{1 \leq r \leq m; 1 \leq t \leq n} s_{[r,t]}^{2(r,t)j_r(\alpha)j_t(\beta)}.$$

Proof. The fact that $m_{m,n}(x, y) = Z(A, (1 + 2x + y)^{1/2})$ follows from (5.4.4) in [5]. Thus, we need only show that the formula for $Z(A)$ is correct.

Suppose that the vertex set of $K_{m,n}$ is $V = V_1 \cup V_2$, where $V_1 = \{a_1, ..., a_m\}$ is the set of vertices of one color and $V_2 = \{b_1, ..., b_n\}$ is the set of vertices of the other color.

Notice that X contains $2 \cdot m \cdot n$ elements. By Definition 3.2,

$$Z(A) = Z(A; s_1, ..., s_{2mn}) = \frac{1}{|A|} \sum_{(\alpha,\beta) \in A} \prod_{1 \leq k \leq m \cdot n} s_k^{j_k((\alpha,\beta))},$$

where for each $1 \leq k \leq n$ and for each $(\alpha, \beta) \in A$, $j_k((\alpha, \beta))$ is the number of cycles of length k in the cycle decomposition of (α, β).

The cycle structure of any $(\alpha, \beta) \in A$ is completely determined by the cycle structures of α and β. That is, the correspondence between the permutations in $S_m \times S_n$ and A induces a correspondence between the contribution of $\phi = (\alpha, \beta)$ to $Z(S_m \times S_n)$ and the contribution of $\phi' = (\alpha, \beta)$ to $Z(A)$.

Let $\phi = (\alpha, \beta) \in S_m \times S_n$, whose contribution to $Z(S_m \times S_n)$ is $\prod s_k^{j_k(\phi)}$. As in the proof of Theorem 3.6, there is no contribution made by ϕ' to the corresponding term in $Z(A)$ from elements of the form $(x, y) \in X$, where x and y are both contained in a single cycle of ϕ. The only contribution comes from elements where x and y are contained in different cycles of ϕ.

To calculate this contribution, let $(x, y) \in X$ such that x and y are in different cycles of ϕ. W.l.o.g., suppose that $x \in V_1$ and $y \in V_2$. Let z_r be the cycle of length r containing x, and let z_t be the cycle of length t containing y. Then z_r and z_t induce a cycle of length $[r, t]$ on (x, y). There are $2rt$ ordered pairs which are affected by the two cycles. Thus, there are $((2rt)/[r, t]) = 2(r, t)$ cycles induced by z_r and z_t.

Since $|A| = m!n!$, then the above formula becomes

$$
\begin{aligned}
Z(A; s_1, ..., s_{m \cdot n}) &= \frac{1}{m!n!} \sum_{(\alpha,\beta) \in A} \prod_{1 \leq k \leq m \cdot n} s_k^{j_k((\alpha,\beta))} \\
&= \frac{1}{m!n!} \sum_{(\alpha,\beta) \in A} \prod_{1 \leq r \leq m; 1 \leq t \leq n} s_{[r,t]}^{2(r,t)j_r(\alpha)j_t(\beta)},
\end{aligned}
$$

where $j_r(\alpha)$ is the number of cycles of length r in the cycle decomposition of α and $j_t(\beta)$ is the number of cycles of length t in the cycle decomposition of β. $\square$

Notation 3.24. *We will frequently use $m_{m,n}$ to denote $m_{m,n}(1,1)$, the number of (m,n) mixed bipartite graphs up to isomorphism.*

As was the case for Theorem 3.6, the above theorem holds for $m = n$ only if the colors are not interchangeable. However, as mentioned before, this is precisely the case in which we are interested since the graphs associated with precodes have directed edges.

4. COUNTING CODES

Definition 4.1. *Let $m, n \in \mathbb{Z}^+$. We let $p_{m,n}$ denote the number of (m, n) precodes up to isomorphism and $c_{m,n}$ denote the number of (m, n) codes up to isomorphism.*

In this section, we show how to compute $c_{m,n}$ using the results from Section 3. Recall from Definition A.2 and Remark 2.6 that if $\mathcal{A}$ is a code, then each of the columns in its associated matrix, $M_{\mathcal{A}}$, is of one of the four types: s, e, d, and o.

Definition 4.2. *A code is said to be of type O (or an O code) if all of the columns in its matrix representation are o columns. A code is said to be of type E (or an E code) if its matrix representation has at least one e column and all of the columns in its matrix representation are e or o columns. A code is said to be a strictly E code if all of the columns in its matrix representation are e columns. The definitions for D, strictly D, S, and strictly S codes are analogous.*

Notation 4.3. *We use $e_{m,n}$ and $eStr_{m,n}$ to denote the number of E and strictly E (m, n) codes, respectively, up to isomorphism. The definitions of $o_{m,n}$, $s_{m,n}$, $d_{m,n}$, $sStr_{m,n}$, and $dStr_{m,n}$ are analogous.*

We begin by counting the number of (m, n) O, S, E, and D codes up to isomorphism.

4.1. *O* **Codes.** We begin with the trivial codes.

Lemma 4.4. $o_{m,n} = 1.$

Proof. The only (m, n) O code is the code with no edges. $\qquad\square$

4.2. *S* **Codes.** Recall that these are the non-trivial self-companion codes.

Lemma 4.5. *$sStr_{m,n}$ is the number of partitions of n into m nonnegative parts; that is, the number of partitions of n into m or fewer parts. Furthermore, $s_{m,n} = \sum_{1 \le j \le n} sStr_{m,j}$.*

Proof. Let $\mathcal{A}$ be an (m,n) S code with $P_{\mathcal{A}} = \{\pi_1, ..., \pi_m\}$. For any $\kappa \in C_{\mathcal{A}}$, the set of edges in $\mathcal{A}$ incident on κ is either empty or is of the form $\{(\pi_j, \kappa), (\kappa, \pi_j)\}$ for some $\pi_j \in P_{\mathcal{A}}$. For each $1 \leq j \leq m$, let n_j be the number of codetext vertices κ for which $\{(\pi_j, \kappa)\}$ is an edge in $\mathcal{A}$. Since we are only concerned with counting codes up to isomorphism, then by renaming if necessary, we may assume that $n_1 \leq ... \leq n_m$. Thus, each (m,n) S code can be represented by the ordered m-tuple $(n_1, ..., n_m)$, and it is clear that two (m,n) S codes are isomorphic if and only if they are represented by the same m-tuple. If $\mathcal{A}$ is a strictly S code, then $n = \sum_{1 \leq j \leq m} n_j$. This proves the first statement.

Now, suppose that $\mathcal{A}$ is an S code with precisely k isolated codetext vertices. That is, suppose there are exactly k o-columns in $M_{\mathcal{A}}$, so that $\sum_{1 \leq j \leq m} n_j = n - k$. Thus, the ordered m-tuple $(n_1, ..., n_m)$ which represents $\mathcal{A}$ is also the m-tuple used to represent one of the isomorphism classes of strictly S $(m, n-k)$ codes. Thus, there is a one-to-one correspondence between the isomorphism classes of (m,n) S codes with exactly k isolated codetext vertices and the isomorphism classes of strictly S $(m, n-k)$ codes. This proves the lemma. $\qquad\square$

4.3. E and D Codes. The E codes (and strictly E codes) have nonempty encode relations and empty decode relations. There is a one-to-one correspondence between the E codes and D codes given via $\mathcal{A} = (P_{\mathcal{A}}, C_{\mathcal{A}}, e_{\mathcal{A}}, \emptyset) \leftrightarrow \mathcal{A}^{pn}$. Furthermore, this correspondence preserves isomorphism . Thus, $d_{m,n} = e_{m,n}$.

For the purpose of counting, the E codes can be represented by bipartite graphs. Recall that bipartite graphs are precisely those graphs that can be bicolored; that is, their vertex set can be partitioned into two disjoint, nonempty subsets such that no two adjacent vertices are contained in the same subset. In our case, the two sets are $P_{\mathcal{A}}$ and $C_{\mathcal{A}}$. The edge set of the bipartite graph is

$$\{\{\pi, \kappa\} | (\pi, \kappa) \in e_{\mathcal{A}}\}$$

Furthermore, if $m \neq n$, it is clear that two (m,n) E codes are isomorphic if and only if their corresponding bipartite graphs are isomorphic. If $m = n$, we must be a bit more careful since

31

precodes are represented by digraphs. Recall that in Example 3.7, we constructed two codes which are not isomorphic, but which have isomorphic corresponding bipartite graphs (if we are allowed to interchange the colors). Thus, for any m and n, counting the number of (m,n) E codes up to isomorphism is equivalent to counting the number of (m,n) bipartite graphs, *where the colors cannot be interchanged.*

Lemma 4.6. $d_{m,n} = e_{m,n} = b_{m,n} - 1$ *and* $eStr_{m,n} = bStr_{m,n}$, *where* $b_{m,n}$ *and* $bStr_{m,n}$ *are as defined in Remark 3.13 and Definition 3.18.*

Proof. The proof that $d_{m,n} = e_{m,n} = b_{m,n} - 1$ was given in the discussion preceeding the lemma, keeping in mind that $b_{m,n}$ counts the single O code (the code with no edges). Thus, to obtain $e_{m,n}$, we must subtract 1 from $b_{m,n}$.

As in Definition 3.18, $bStr_{m,n}$ denotes the number of 2-strict (m,n) bipartite graphs up to isomorphism. Recall that a bipartite graph is said to be 2-strict if none of the n vertices of the second color are isolated. However, an (m,n) code is a strictly E code if all of the columns in its matrix representation are e-columns. That is, if none of the n codetext vertices in the associated bipartite graph are isolated. Thus, $eStr_{m,n}$ is also the number of 2-strict (m,n) bipartite graphs up to isomorphism. □

4.4. **Integration.** It might seem that the number of isomorphism classes of codes is

$$\sum_{(n_1,n_2,n_3,n_4)\in ncomp(n)} (o_{m,n_1} s_{m,n_2} e_{m,n_3} d_{m,n_4}),$$

where $ncomp(n)$ is the set of all nonnegative compositions of n (see Definition 3.8) and the factors are defined to be 1 if the corresponding term of the composition is 0. However, this is not the case. Although each of the columns in the matrix representation of a code is of one of the four types, the following example illustrates why we cannot count as we might have hoped.

Example 4.7. *Let* $m = 2 = n$, *and consider the composition* $(0,1,1,0)$ *of* n *which represents the codes with no o columns, one s column, one e column, and no d columns. We have that* $o_{m,0} = 1$,

$s_{m,1} = 1$, $e_{m,1} = 1$, and $d_{m,0} = 1$. However, there are two nonisomorphic $(2,2)$ codes corresponding to the composition $(0,1,1,0)$. They are depicted as Precodes A and B in Figure 2.
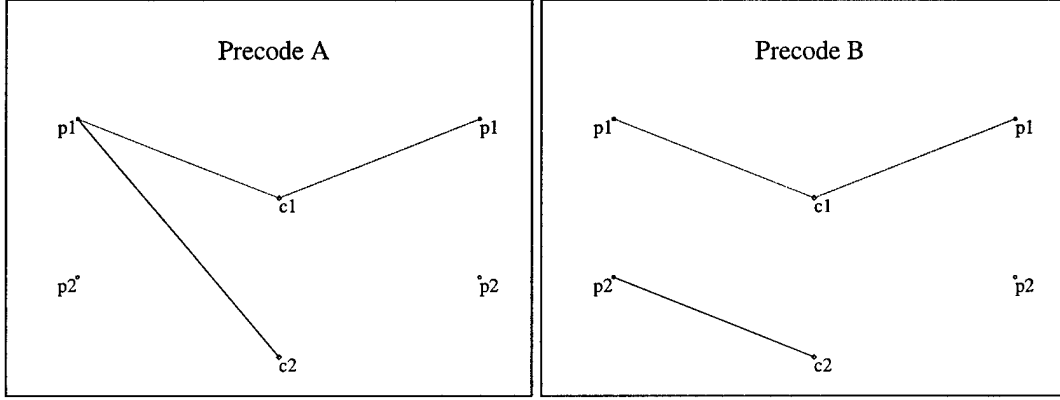
FIGURE 2. Two Nonisomorphic Codes with the Same Partition

We need to take another approach. We begin with an extension of Definition 4.2.

Definition 4.8. *A code is said to be an SE code if its matrix representation has at least one s column, at least one e column, and no d columns. A code is said to be a strictly SE code if it is an SE code with no o columns. There are analogous definitions for SD, strictly SD, ED, and strictly ED codes. A code is said to be of type SED if its matrix representation has at least one s column, at least one e column, and at least one d column.*

Notation 4.9. *We use $se_{m,n}$ and $seStr_{m,n}$ to denote the number of SE and strictly SE (m,n) codes, respectively, up to isomorphism. The definitions of $sd_{m,n}$, $sdStr_{m,n}$, $ed_{m,n}$, $edStr_{m,n}$, and $sed_{m,n}$ are analogous.*

There is a one-to-one correspondence between the SE codes and SD codes given via $\mathcal{A} \leftrightarrow \mathcal{A}^{pn}$. Furthermore, this correspondence preserves isomorphism. Thus, $sd_{m,n} = se_{m,n}$ and $sdStr_{m,n} = seStr_{m,n}$. We are now ready to state the main theorem of this section.

Theorem 4.10.

$$c_{m,n} = o_{m,n} + s_{m,n} + e_{m,n} + d_{m,n} + se_{m,n} + sd_{m,n} + ed_{m,n} + sed_{m,n}$$

$$= o_{m,n} + s_{m,n} + 2 \cdot e_{m,n} + 2 \cdot se_{m,n} + ed_{m,n} + sed_{m,n}.$$

Proof. This is clear since each (m, n) code is of precisely one of the following types: O, S, E, D, SE, SD, ED, and SED. $\square$

We have already computed $o_{m,n}$, $s_{m,n}$, $e_{m,n}$, and $d_{m,n}$. It therefore remains to compute $ed_{m,n}$, $se_{m,n}$, $sd_{m,n}$, and $sed_{m,n}$.

4.5. *ED* **Codes.** Recall that these are the non-trivial codes with no s codetext elements.

Lemma 4.11.

$$ed_{m,n} = \begin{cases} 0, & \text{if } n < 2 \\ \sum_{0 \le j \le (n-2)} \sum_{1 \le k \le (n-j-1)} (eStr_{m,k} \cdot eStr_{m,n-j-k}), & \text{if } n \ge 2. \end{cases}$$

Proof. Since an *ED* code must have at least one e codetext element and at least one d codetext element, then $ed_{m,n} = 0$ if $n < 2$.

For $n \ge 2$, the number of isomorphism classes of *ED* codes is $\sum_j \sum_k I(k, n - (j + k))$, where j represents the number of o-columns in an *ED* code, k is the number of e-columns, $n - (j + k)$ is the number of d-columns, and $I(k, n - (j + k))$ is the number of isomorphism classes of *ED* codes with exactly k e-columns and $n - (j + k)$ d-columns. But, $I(k, n - (j + k)) = eStr_{m,k} \cdot eStr_{m,n-(j+k)}$, since $eStr_{m,k}$ is the number of (m, k) strictly E codes and $eStr_{m,n-(j+k)}$ is the number of $(m, n - (j + k))$ strictly D codes. Since there must be at least one e and at least one d column in the matrix representation of any *ED* code, then j can be no larger than $n - 2$. Furthermore, k must be at least 1 and no more than $n - (j + 1)$.

Notice that when $n < 2$, there is no j satisfying $0 \le j \le (n - 2)$. Thus, we may employ the standard practice of defining $\sum_{0 \le j \le (n-2)} \sum_{1 \le k \le (n-j-1)} (eStr_{m,k} \cdot eStr_{m,n-j-k})$ to be 0 in this case. Thus, the above formula actually holds for all $n \in \mathbb{Z}^+$. $\square$

4.6. *SE* and *SD* Codes. As noted above, there is a one-to-one correspondence between the *SE* codes and *SD* codes which preserves isomorphism. Thus, $se_{m,n} = sd_{m,n}$.

Notation 4.12. *For a given partition J of $q \in \mathbb{Z}^+$, we let $|J|$ denote the number of parts in J. For example, if $q = 4$ and $J = (1,1,2)$, then $|J| = 3$.*

Now, let J be a partition of some $s \in \mathbb{Z}^+$ such that $s \le n$ and $|J| \le m$. Let $M(J)$ denote the partition of $|J|$ which tracks the number of times each value in J appears. For example, suppose $m = 4$, $n = 5$, $s = 4$, and $J = (1,1,2)$. Then $M(J) = (2,1)$, indicating that the value 1 appears two times and the value 2 appears one time. Finally, we let $K(J)$ denote the partition of m formed by adding at most one term to $M(J)$. In the above example, $K(J) = (2,1,1)$.

Lemma 4.13.

$$sd_{m,n} = se_{m,n} = \begin{cases} 0, & \text{if } n < 2 \\ \\ \sum_{1 \le c \le (n-1)} \sum_{J}(b_{m,c,K(J)} - 1), & \text{if } n \ge 2, \end{cases}$$

where the second sum is over all partitions J of $s = n - c$ such that $|J| \le m$ and where $K(J)$ is as defined in Notation 4.12. Furthermore, $sdStr_{m,n} = seStr_{m,n} = se_{m,n} - \sum_{1 \le k \le (n-1)} seStr_{m,n-k}$.

Proof. Since an *SE* code must have at least one s codetext element and at least one e codetext element, then $se_{m,n} = 0$ if $n < 2$.

Now, let c represent the sum of the number of e and o codetext elements in an *SE* code, i.e., $c = n - s$, where s is the number of s codetext elements. Since the code must have at least one s column, $c \le n - 1$. Since there must be at least one e column, $c \ge 1$.

Two *SE* codes with nonisomorphic underlying S codes (recall Definition 2.16) are clearly not isomorphic. As we saw in Lemma 4.5, there is a one-to-one correspondence between the isomorphism classes of S codes with s codetext elements of type s and the partitions of s with m or fewer parts. Each entry in such a partition represents a connected component of the S code associated with the partition.

To finish the proof, we need to determine the number of isomorphism classes of SE codes whose underlying S code is associated with a given partition J of s such that $|J| \leq m$. So, let $J = (n_1, ..., n_k)$ be a partition of s with $k \leq m$ and $n_1 \leq ... \leq n_k$. As in Lemma 4.5, we let S' be the (m, n) S code associated with J. In particular,

$$S' = (P_{S'} = \{\pi_1, ..., \pi_m\}, C_{S'} = \bigcup_{1 \leq j \leq (k+1)} C_j, e_{S'} = \bigcup_{1 \leq j \leq k} (\{\pi_j\} \times C_j), d_{S'} = e_{S'}^{\leftarrow}),$$

where $C_1, ..., C_k, C_{k+1}$ are pairwise disjoint subsets of $C_{S'}$ such that for each $1 \leq j \leq k$, $|C_j| = n_j$ and such that $|C_{k+1}| = n - s = c$. For each $1 \leq j \leq k$, $S_j' = (\{\pi_j\}, C_j, \{\pi_j\} \times C_j, C_j \times \{\pi_j\})$ is an s component of S'.

Now, any SE code whose underlying S code is isomorphic to S' is isomorphic to an SE code of the form $S' \cup E' = (P_{S'}, C_{S'}, e_{S'} \cup e', d_{S'})$ for some E code $E' = (P_{S'}, C_{S'}, e', \emptyset)$, where $e_{S'}$ and e' are disjoint. We must determine the number of codes of this type up to isomorphism. We are free to construct e' using as edges any of the elements in $P_{S'} \times C_{k+1}$, but we may not use any elements in $P_{S'} \times (\bigcup_{1 \leq j \leq k} C_j)$, since $S' \cup E'$ would fail to be a code.

As above, we let $M(J)$ denote the partition of $|J|$ which tracks the number of times each value in J appears. Then $M(J)$ contains one entry for each connected s component in S' up to isomorphism, and the value of each entry is the number of s components in the corresponding isomorphism class. Now, $|J|$ is the number of s components of S' and therefore also the number of plaintext elements which are part of some s component. As above, we let $K(J)$ denote the partition of m formed by tacking at most one term on to $M(J)$. This term represents the number of plaintext vertices which are not part of any s component of S'.

If $E_1 = (P_{S'}, C_{k+1}, e_1, \emptyset)$ and $E_2 = (P_{S'}, C_{k+1}, e_2, \emptyset)$, then $S' \cup E_1$ and $S' \cup E_2$ are isomorphic if and only if the bipartite graphs corresponding to E_1 and E_2 are equivalent with respect to $K(J)$, where this equivalence is as defined in Definition 3.10. The number of (m, c) E codes up to $K(J)$-equivalence is precisely $b_{m,c,K(J)} - 1$, where $b_{m,c,K(J)}$ is as in Notation 3.13. Note that we must

subtract 1 from $b_{m,c,K(J)}$ since we must not count the code with no edges. This proves the first formula.

The number of (m,n) strictly SE codes up to isomorphism is the number of (m,n) SE codes up to isomorphism minus the number of $(m, n-k)$ strictly SE codes up to isomorphism, where k runs from 1 to $n-1$. Here, k represents the number of o columns we have. We do not let k run to n since the single O code is not counted in $se_{m,n}$. $\square$

4.7. SED Codes. Recall that in Corollary 3.19, we define $bStr_{p,q,M}$ to be the number of 2-strict (p,q) bipartite graphs up to M-equivalence, where $p, q \in \mathbb{Z}^+$ and M is a partition of p.

Lemma 4.14.

$$
sed_{m,n} = \begin{cases} 0, & \text{if } n < 3 \\ \sum_{1 \le c \le (n-1)} \sum_{J} \sum_{1 \le e \le (n-s-1)} \sum_{1 \le d \le (n-s-e)} (bStr_{m,e,K(J)} \cdot bStr_{m,d,K(J)}), & \text{if } n \ge 3, \end{cases}
$$

where the second sum is over all partitions J of $s = n - c$ such that $|J| \le m$ and where $K(J)$ is as defined in Notation 4.12.

Proof. Since an SED code must have at least one s codetext vertex, at least one e codetext vertex, and at least one d codetext vertex, then the number of SED codes is 0 if $n < 3$.

Let c represent the sum of the number of e and o codetext elements in an SED code, i.e., $c = n - s$, where s is the number of s codetext elements. Since the code must have at least one s column, $c \le n - 1$. Since there must be at least one e column, $c \ge 1$.

The justification for the structure of the first two sums is the same as that given in the proof of Lemma 4.13. To finish the proof, we need to show that the number of isomorphism classes of SED codes whose underlying S code is associated with a given partition J of s such that $|J| \le m$ is

$$
\sum_{1 \le e \le (n-s-1)} \sum_{1 \le d \le (n-s-e)} (bStr_{m,e,K(J)} \cdot bStr_{m,d,K(J)}).
$$

So, let $J = (n_1, ..., n_k)$ be a partition of s with $k \leq m$ and $n_1 \leq ... \leq n_k$. As in Lemma 4.5, we let S' be the (m, n) S code associated with J. In particular,

$$S' = (P_{S'} = \{\pi_1, ..., \pi_m\}, C_{S'} = \bigcup_{1 \leq j \leq (k+1)} C_j, e_{S'} = \bigcup_{1 \leq j \leq k} (\{\pi_j\} \times C_j), d_{S'} = e_{S'}^{\leftarrow}),$$

where $C_1, ..., C_k, C_{k+1}$ are pairwise disjoint subsets of $C_{S'}$ such that for each $1 \leq j \leq k$, $|C_j| = n_j$ and such that $|C_{k+1}| = n - s = c$. For each $1 \leq j \leq k$, $S_j' = (\{\pi_j\}, C_j, \{\pi_j\} \times C_j, C_j \times \{\pi_j\})$ is an s component of S'.

Let e represent the number of e codetext elements in an SED code whose underlying S code is isomorphic to S'. Since we must have at least one e and at least one d codetext element, then $1 \leq e \leq n - (s + 1)$. Let d represent the number of d codetext elements in such a code. Then $d \leq n - (s + e)$. This justifies the form of the last two sums.

Finally, we must show that the number of isomorphism classes of SED codes whose underlying S code is S' and which have e columns of type e and d columns of type d is $(bStr_{m,e,K(J)} \cdot bStr_{m,d,K(J)})$.

Recall that C_{k+1} is the set of $n - s$ codetext elements which are not part of any of the s components of S'. Let C_e, C_d, and C_o be pairwise disjoint subsets of C_{k+1} such that $|C_e| = e$, $|C_d| = d$, $C_o = n - (s + e + d)$. That is, $C_{k+1} = C_e \cup C_d \cup C_o$. Any SED code whose underlying S code is isomorphic to S' and which has e columns of type e and d columns of type d is isomorphic to an SED code of the form $S' \cup E' \cup D' = (P_{S'}, C_{S'}, e_{S'} \cup e', d_{S'} \cup d')$ for some strictly E code $E' = (P_{S'}, C_e, e', \emptyset)$ and some strictly D code $D' = (P_{S'}, C_d, \emptyset, d')$.

As above, we let $M(J)$ denote the partition of $|J|$ which tracks the number of times each value in J appears. Then $M(J)$ contains one entry for each connected s component in S' up to isomorphism, and the value of each entry is the number of s components in the corresponding isomorphism class. Now, $|J|$ is the number of s components of S' and therefore also the number of plaintext elements which are part of some s component. As above, we let $K(J)$ denote the partition of m formed by tacking at most one term on to $M(J)$. This term represents the number of plaintext vertices which are not part of any s component of S'.

For strict codes $E_1 = (P_{S'}, C_e, e_1, \emptyset), E_2 = (P_{S'}, C_e, e_2, \emptyset), D_1 = (P_{S'}, C_d, \emptyset, d_1)$, and $D_2 = (P_{S'}, C_d, \emptyset, d_2)$, then $S' \cup E_1 \cup D_1$ and $S' \cup E_2 \cup D_2$ are isomorphic if and only if the bipartite graphs corresponding to E_1 and E_2 are $K(J)$-equivalent and the bipartite graphs corresponding to D_1 and D_2 are $K(J)$-equivalent, where this equivalence is as defined in Definition 3.10. The number of strictly E codes of the form $E' = (P_{S'}, C_e, e', \emptyset)$ up to $K(J)$-equivalence is $bStr_{m,e,K(J)}$, and the number of strictly D codes of the form $D' = (P_{S'}, C_d, \emptyset, d')$ up to $K(J)$-equivalence is $bStr_{m,d,K(J)}$. This proves the lemma. $\qquad\Box$

Corollary 4.15. *Let* $S = se_{m,n} + sd_{m,n} + sed_{m,n}$. *Then*

$$
S = \begin{cases}
0, & \text{if } n < 2 \\[2ex]
\sum_{1 \le c \le (n-1)} \sum_J (2 \cdot (b_{m,c,K(J)} - 1) \\[1ex]
\quad + \sum_{1 \le e \le (n-s-1)} \sum_{1 \le d \le (n-s-e)} (bStr_{m,e,K(J)} \cdot bStr_{m,d,K(J)})), & \text{if } n \ge 2,
\end{cases}
$$

where the second sum is over all partitions J of $s = n - c$ such that $|J| \le m$ and where $K(J)$ is as defined in Notation 4.12.

Proof. This is just the combination of Lemmas 4.13 and 4.14. $\qquad\Box$

4.8. Self-companion Codes.

The self-companion (m, n) codes are precisely the (m, n) S codes as defined in Definition 4.2, along with the single (m, n) O code. The next theorem follows immediately.

Theorem 4.16. *Let $sc_{m,n}$ denote the number of self-companion (m, n) codes up to isomorphism. Then $sc_{m,n} = s_{m,n} + 1$, where $s_{m,n}$ is as given in Lemma 4.5.*

4.9. Janiform Codes.

Recall that a janiform code is a code whose opposite is also a code.

Notation 4.17. *Let m and n be nonnegative integers. We let $S_{m,m}$ denote the number of (m, m) strictly S codes whose opposites are also strictly S codes. We let $E_{m,n}$ denote the number of (m, n) strictly E codes whose opposites are strictly D codes. We define $D_{m,n}$ analogously. Furthermore,*

if $m = 0$ *and* $n = 0$, *we set* $S_{m,n} = E_{m,n} = D_{m,n} = 1$, *representing the empty code* $(\emptyset, \emptyset, \emptyset, \emptyset)$. *We note that in the theorems below, we will encounter neither the case* $m = 0$ *and* $n \neq 0$ *nor the case* $m \neq 0$ *and* $n = 0$.

Lemma 4.18. $S_{m,m} = 1$ *for any nonnegative integer* m.

Proof. By definition, $S_{0,0} = 1$. Let $m \in \mathbb{Z}^+$, and let

$$S = (P_S = \{\pi_1, ..., \pi_m\}, C_S = \{\kappa_1, ..., \kappa_m\}, e_S, d_S)$$

be an (m, m) strictly S code whose opposite is also a strictly S code. Let S' be a connected s component of S. Note that if either the encode or decode relation of S' contained more than one edge, then S^{op} would not be a code. Thus, $S' = (\{\pi_i\}, \{\kappa_j\}, \{(\pi_i, \kappa_j)\}, \{(\kappa_j, \pi_i)\})$ for some $\pi_i \in P_S$ and some $\kappa_j \in C_S$. By renaming if necessary, we may assume that $S' = (\{\pi_j\}, \{\kappa_j\}, \{(\pi_j, \kappa_j)\}, \{(\kappa_j, \pi_j)\})$. Since S and S^{op} are strictly S codes, then S has no isolated plaintext or codetext elements. Thus, there are precisely m s components, each of the above form. There is only one such code up to isomorphism. $\square$

Lemma 4.19. *Let* $m, n \in \mathbb{Z}^+$. *Then* $E_{m,n} = D_{m,n} = B_{m,n}$, *where* $B_{m,n}$ *is as in Definition 3.18.*

Proof. Notice that any (m, n) E code is janiform. Thus, following the proof of Lemma 4.6, the number of strictly E (m, n) codes whose opposites are strictly D codes is the number of (m, n) bipartite graphs which have no isolated vertices up to isomorphism. However, this is precisely $B_{m,n}$. Furthermore, there is the obvious one-to-one correspondence between (m, n) E codes and (m, n) D codes which preserves isomorphism. Thus, $E_{m,n} = D_{m,n}$. $\square$

Theorem 4.20. *Let* $m, n \in \mathbb{Z}^+$, *and let* $j_{m,n}$ *denote the number of* (m, n) *janiform codes up to isomorphism. Then*

$$j_{m,n} = \sum_{M,N} E_{m_e, n_e} E_{m_d, n_d},$$

where the sum is taken over all nonnegative compositions $M = (m_s, m_e, m_d, m_o)$ and $N = (n_s = m_s, n_e, n_d, n_o)$ of m and n, respectively, such that m_a and n_a are either both zero or both nonzero for each $a \in \{s, e, d\}$.

Proof. Let $\mathcal{A} = (P_{\mathcal{A}} = \{\pi_1, ..., \pi_m\}, C_{\mathcal{A}} = \{\kappa_1, ..., \kappa_n\}, e_{\mathcal{A}}, d_{\mathcal{A}})$ be an (m, n) janiform code with synoptic codebook matrix $M_{\mathcal{A}}$. Recall that $M_{\mathcal{A}^{op}}$ is formed from $M_{\mathcal{A}}{}^t$ by interchanging the bits in each entry. That is, $M_{\mathcal{A}^{op}}$ is defined via

$$
M_{\mathcal{A}^{op}}(\kappa, \pi) = \begin{cases} s \text{ if } M_{\mathcal{A}}{}^t(\kappa, \pi) = s, \\ e \text{ if } M_{\mathcal{A}}{}^t(\kappa, \pi) = d, \\ d \text{ if } M_{\mathcal{A}}{}^t(\kappa, \pi) = e, \\ o \text{ if } M_{\mathcal{A}}{}^t(\kappa, \pi) = o. \end{cases}
$$

Since $\mathcal{A}^{op}$ is a code, this implies that each of the rows of $M_{\mathcal{A}}$ (as well as each of the columns of $M_{\mathcal{A}}$) is of one of the four types: s, e, d, and o. Let P_s and C_s be the sets of plaintext and codetext elements, respectively, of type s. Let $m_s = |P_s|$ and $n_s = |C_s|$. Let P_e, C_e, P_d, C_d, P_o, and C_o be defined analogously, with cardinalities m_e, n_e, m_d, n_d, m_o, and n_o, respectively. We note that if $\alpha \in \{s, e, d\}$, then either m_α and n_α are both zero or both nonzero. Since e and d columns (or rows) may contain o entries, then it is not necessarily the case that either m_o and n_o are both zero or both nonzero. Since each row and column of type s in $M_{\mathcal{A}}$ has exactly one s entry, then $m_s = n_s$.

We have that $P_{\mathcal{A}} = P_s \cup P_e \cup P_d \cup P_o$ and $C_{\mathcal{A}} = C_s \cup C_e \cup C_d \cup C_o$ are disjoint unions. Let $e_s = (P_s \times C_s) \cap e_{\mathcal{A}}$, $e_e = (P_e \times C_e) \cap e_{\mathcal{A}}$, $d_s = (C_s \times P_s) \cap d_{\mathcal{A}}$, and $d_d = (C_d \times P_d) \cap d_{\mathcal{A}}$. It is clear that $\mathcal{A} = S \cup E \cup D \cup O = (P_{\mathcal{A}}, C_{\mathcal{A}}, e_s \cup e_e, d_s \cup d_d)$, where $S = (P_s, C_s, e_s, d_s)$, $E = (P_e, C_e, e_e, \emptyset)$, $D = (P_d, C_d, \emptyset, d_d)$, and $O = (P_o, C_o, \emptyset, \emptyset)$. Notice that S is an (m_s, m_s) strictly S code. Also, E is a strictly E code, and E^{op} is a strictly D code. Similarly, D is a strictly D code, and D^{op} is a strictly E code. Finally, O is the unique (m_o, n_o) O code.

Let $\mathcal{A}_1 = S_1 \cup E_1 \cup D_1 \cup O_1$ and $\mathcal{A}_2 = S_2 \cup E_2 \cup D_2 \cup O_2$ be janiform codes which are both associated with the nonnegative compositions $M = (m_s, m_e, m_d, m_o)$ and $N = (n_s = m_s, n_e, n_d, n_o)$

of m and n, respectively, given above. That is, S_1 and S_2 are (m_s, m_s) strictly S codes, E_1 and E_2 are (m_e, n_e) strictly E codes whose opposites are strictly D codes, D_1 and D_2 are (m_d, n_d) strictly D codes whose opposites are strictly E codes, and $O_1 = O_2$ is the unique (m_o, n_o) O code.

It is clear that $\mathcal{A}_1$ and $\mathcal{A}_2$ are isomorphic if and only if their component codes are isomorphic. This proves that the number of (m, n) janiform codes up to isomorphism is given by $\sum_{M,N} S_{m_s, m_s} E_{m_e, n_e} D_{m_d, n_d}$, where the sum is taken over all appropriate nonnegative compositions

$$M = (m_s, m_e, m_d, m_o) \text{ and } N = (n_s = m_s, n_e, n_d, n_o)$$

of m and n, respectively. However, by Lemmas 4.18 and 4.19, this is $\sum_{M,N} E_{m_e, n_e} E_{m_d, n_d}$. $\quad\square$

4.10. Self-opposite Codes. Recall first that a janiform code is a code whose opposite is also a code. Thus, a self-opposite code is janiform. Furthermore, if $\mathcal{A}$ is self-opposite, then $\mathcal{A} = (P_\mathcal{A}, P_\mathcal{A}, e_\mathcal{A}, e_\mathcal{A})$.

Theorem 4.21. *Let $m \in \mathbb{Z}^+$, and let $sop_{m,m}$ denote the number of (m, m) self-opposite codes up to isomorphism. Then*

$$sop_{m,m} = \sum_M E_{m_e, m_d},$$

where the sum is taken over all nonnegative compositions $M = (m_s, m_e, m_d, m_o)$ of m such that m_e and m_d are either both zero or both nonzero.

Proof. Let $\mathcal{A} = (P_\mathcal{A} = \{\pi_1, ..., \pi_m\}, C_\mathcal{A} = P_\mathcal{A}, e_\mathcal{A}, d_\mathcal{A} = e_\mathcal{A})$ be an (m, m) self-opposite code. We let the sets P_s, C_s, P_e, C_e, P_d, C_d, P_o, C_o and their associated cardinalities m_s, n_s, m_e, n_e, m_d, n_d, m_o, n_o, respectively, be defined as in the proof of Theorem 4.20. We also define e_s, e_e, d_s, and d_d as in the proof. We have then that $\mathcal{A} = S \cup E \cup D \cup O = (P_\mathcal{A}, C_\mathcal{A}, e_s \cup e_e, d_s \cup d_d)$, where $S = (P_s, C_s, e_s, d_s)$, $E = (P_e, C_e, e_e, \emptyset)$, $D = (P_d, C_d, \emptyset, d_d)$, and $O = (P_o, C_o, \emptyset, \emptyset)$. We saw that S is an (m_s, m_s) strictly S code, E and D^{op} are strictly E codes, D and E^{op} are strictly D codes, and O is the unique (m_o, n_o) O code.

Since $\mathcal{A}$ is self-opposite, then $P_s = C_s$ and $e_s = d_s = 1_{P_s}$, so that $S = (P_s, P_s, 1_{P_s}, 1_{P_s})$. Furthermore, $d_d = e_e$, $P_d = C_e$, and $C_d = P_e$ so that $D = (C_e, P_e, \emptyset, e_e)$. Hence, the structure of D is completely determined by the structure of E. Since $m_s = n_s$, $m_e = n_d$, $m_d = n_e$, and $m_s + m_e + m_d + m_o = m = n_s + n_e + n_d + n_o$, then $m_o = n_o$. Thus, we can associate $\mathcal{A}$ with the single nonnegative composition $M = (m_s, m_e, m_d, m_o)$ of m. We further note that as in Theorem 4.20, either m_e and n_e are both zero or both nonzero. Since $m_d = n_e$, then either m_e and m_d are both zero or both nonzero.

Let $\mathcal{A}_1 = S_1 \cup E_1 \cup D_1 \cup O_1$ and $\mathcal{A}_2 = S_2 \cup E_2 \cup D_2 \cup O_2$ be self-opposite codes which are both associated with the nonnegative composition $M = (m_s, m_e, m_d, m_o)$ of m given above. It is clear that $\mathcal{A}_1$ and $\mathcal{A}_2$ are isomorphic if and only if their component codes are isomorphic. Since the structure of D_1 is completely determined by that of E_1 (and similarly for D_2 and E_2), then the number of (m, m) self-opposite codes up to isomorphism is $\sum_M S_{m_s, m_s} E_{m_e, m_d}$, where the sum is taken over all appropriate nonnegative compositions $M = (m_s, m_e, m_d, m_o)$ of m. By Lemma 4.18, this is $\sum_M B_{m_e, m_d}$. $\qquad\qquad\square$

4.11. **Calculations.** In Appendix D, we provide Maple algorithms which implement the formulae from this section. Some of the results are given in Table 1. The (m, n) entry in the table is of the form

$$
\begin{pmatrix}
o_{m,n} & s_{m,n} & e_{m,n} \\
se_{m,n} & ed_{m,n} & sed_{m,n}
\end{pmatrix}
$$

m/n	1	2	3	4	5
1	1 1 1 0 0 0	1 2 2 1 1 0	1 3 3 3 3 1	1 4 4 6 6 4	1 5 5 10 10 10
2	1 1 2 0 0 0	1 3 6 3 4 0	1 5 12 14 20 9	1 8 21 40 60 58	1 11 33 91 144 224
3	1 1 3 0 0 0	1 3 12 5 9 0	1 6 35 33 63 25	1 10 86 134 282 255	1 15 189 431 1002 1522
4	1 1 4 0 0 0	1 3 21 7 16 0	1 6 86 60 152 49	1 11 316 346 961 694	1 17 1052 1631 5011 6109
5	1 1 5 0 0 0	1 3 33 9 25 0	1 6 189 98 305 81	1 11 1052 785 2649 1525	1 18 5623 5558 20015 18849
6	1 1 6 0 0 0	1 3 49 11 36 0	1 6 385 148 552 121	1 11 3249 1639 6433 2902	1 18 28575 17639 69697 49033
7	1 1 7 0 0 0	1 3 69 13 49 0	1 6 733 213 917 169	1 11 9342 3216 14057 5047	1 18 136757 52750 216919 113942
8	1 1 8 0 0 0	1 3 94 15 64 0	1 6 1323 294 1440 225	1 11 25206 5982 28500 8170	1 18 613893 148910 622016 242189
9	1 1 9 0 0 0	1 3 124 17 81 0	1 6 2283 394 2151 289	1 11 64116 10633 54238 12565	1 18 2583163 397718 1664702 481081
10	1 1 10 0 0 0	1 3 160 19 100 0	1 6 3789 514 3100 361	1 11 155003 18163 98180 18498	1 18 10208742 1007678 4211160 902849

TABLE 1. Enumeration of Codes

5. COUNTING PRECODES

In this section, we show how to compute $p_{m,n}$ (recall Definition 4.1) for $m, n \in \mathbb{Z}^+$. Since there is no standard matrix representation for precodes, one might intuitively conclude that it might be difficult, if not impossible, to count the number of precodes of a given size up to isomorphism. However, it turns out that the task is easier than that of counting codes. To accomplish it, we again turn to the techniques employed in [5].

Theorem 5.1. *For $m, n \in \mathbb{Z}^+$, $p_{m,n} = m_{m,n}$, where $m_{m,n}$ is as defined in Notation 3.24.*

Proof. Any precode $\mathcal{A}$ can be represented by a mixed bipartite graph. We represent an s edge (i.e. an edge in $e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$) by a non-directed edge, and we represent edges in e and d by the appropriate directed edges. If $m \neq n$, it is clear that two (m, n) precodes are isomorphic if and only if their corresponding mixed bipartite graphs are isomorphic. If $m = n$, we must be more careful. Switching the roles of a code's plaintext and codetext elements may result in a code which is not isomorphic to the original, even though the associated mixed graphs are isomorphic. Thus, for any m and n, counting the number of (m, n) precodes up to isomorphism is equivalent to counting the number of (m, n) mixed bipartite graphs, *where the colors cannot be interchanged.* The number of such graphs is $m_{m,n}$. $\square$

5.1. Self-companion Precodes. Recall Definition A.5.

Theorem 5.2. *Let $m, n \in \mathbb{Z}^+$, and let $psc_{m,n}$ denote the number of self-companion (m, n) precodes up to isomorphism. Then*

$$psc_{m,n} = b_{m,n},$$

where $b_{m,n}$ is as given in Notation 3.13.

Proof. Any self-companion precode $\mathcal{A}$ is of the form $\mathcal{A} = (P_{\mathcal{A}}, C_{\mathcal{A}}, e_{\mathcal{A}}, e_{\mathcal{A}}^{\leftarrow})$. Thus,

$$(P_{\mathcal{A}}, C_{\mathcal{A}}, e_{\mathcal{A}}, e_{\mathcal{A}}^{\leftarrow}) \leftrightarrow (P_{\mathcal{A}}, C_{\mathcal{A}}, e_{\mathcal{A}}, \emptyset)$$

gives a one-to-one correspondence (which preserves isomorphism) between the self-companion pre-codes and the precodes whose columns are each of type e or o. These latter precodes are precisely the (m, n) E codes, along with the single (m, n) O code. As in Theorem 4.6, the number of such codes is $e_{m,n} + 1 = b_{m,n}$. □

5.2. Janiform Precodes.

Recall that a janiform precode is a precode whose opposite is a code.

Theorem 5.3. *Let $m, n \in \mathbb{Z}^+$, and let $pj_{m,n}$ denote the number of janiform (m, n) precodes up to isomorphism. Then*

$$pj_{m,n} = c_{m,n},$$

where $c_{m,n}$ is as in Theorem 4.10.

Proof. A one-to-one correspondence between janiform precodes and codes is given via $\mathcal{A} \leftrightarrow \mathcal{A}^{op}$. This correspondence clearly preserves isomorphism. Thus, the number of janiform (m, n) precodes up to isomorphism is the number of (m, n) codes up to isomorphism. □

5.3. Self-opposite Precodes.

Recall Definition A.7.

Theorem 5.4. *Let $m \in \mathbb{Z}^+$, and let $psop_{m,m}$ denote the number of self-opposite (m, m) precodes up to isomorphism. Then*

$$psop_{m,m} = b_{m,m},$$

where $b_{m,m}$ is as given in Notation 3.13.

Proof. Any self-opposite precode $\mathcal{A}$ is of the form $\mathcal{A} = (P_\mathcal{A}, P_\mathcal{A}, e_\mathcal{A}, e_\mathcal{A})$. Thus, $(P_\mathcal{A}, P_\mathcal{A}, e_\mathcal{A}, e_\mathcal{A}) \leftrightarrow (P_\mathcal{A}, P_\mathcal{A}, e_\mathcal{A}, \emptyset)$ gives a one-to-one correspondence (which preserves isomorphism) between the (m, m) self-opposite precodes and the (m, m) precodes whose columns are each of type e or o. These latter precodes are precisely the (m, m) E codes, along with the single (m, m) O code. The number of such codes is $e_{m,m} + 1 = b_{m,m}$. □

5.4. Calculations. In Appendix D, we provide Maple algorithms which implement the formulae from this section. Some of the results are given in Table 2. The (m, n) entry in the table is of the form

$$
\begin{pmatrix}
\dfrac{c_{m,n}}{p_{m,n}} & \dfrac{sc_{m,n}}{psc_{m,n}} \\[2ex]
\dfrac{j_{m,n}}{pj_{m,n}} & \dfrac{sop_{m,n}}{psop_{m,n}}
\end{pmatrix}
$$

m/n	1	2	3	4	5
1	$\frac{4}{4}\;\frac{1}{2}$ $\frac{4}{4}\;\frac{2}{2}$	$\frac{10}{10}\;\frac{2}{3}$ $\frac{6}{10}\;\frac{0}{0}$	$\frac{20}{20}\;\frac{3}{4}$ $\frac{8}{20}\;\frac{0}{0}$	$\frac{35}{35}\;\frac{4}{5}$ $\frac{10}{35}\;\frac{0}{0}$	$\frac{56}{56}\;\frac{5}{6}$ $\frac{12}{56}\;\frac{0}{0}$
2	$\frac{6}{10}\;\frac{1}{3}$ $\frac{6}{6}\;\frac{0}{0}$	$\frac{26}{76}\;\frac{3}{7}$ $\frac{22}{26}\;\frac{5}{7}$	$\frac{87}{420}\;\frac{5}{13}$ $\frac{42}{87}\;\frac{0}{0}$	$\frac{249}{1996}\;\frac{8}{22}$ $\frac{70}{249}\;\frac{0}{0}$	$\frac{628}{7882}\;\frac{11}{34}$ $\frac{106}{628}\;\frac{0}{0}$
3	$\frac{8}{20}\;\frac{1}{4}$ $\frac{8}{8}\;\frac{0}{0}$	$\frac{47}{430}\;\frac{3}{13}$ $\frac{42}{47}\;\frac{0}{0}$	$\frac{231}{8240}\;\frac{6}{36}$ $\frac{124}{231}\;\frac{10}{36}$	$\frac{988}{131505}\;\frac{10}{87}$ $\frac{280}{988}\;\frac{0}{0}$	$\frac{3780}{1757384}\;\frac{15}{190}$ $\frac{568}{3780}\;\frac{0}{0}$
4	$\frac{10}{35}\;\frac{1}{5}$ $\frac{10}{10}\;\frac{0}{0}$	$\frac{76}{1996}\;\frac{3}{22}$ $\frac{70}{76}\;\frac{0}{0}$	$\frac{500}{131505}\;\frac{6}{87}$ $\frac{280}{500}\;\frac{0}{0}$	$\frac{2991}{7880456}\;\frac{11}{317}$ $\frac{928}{2991}\;\frac{20}{317}$	$\frac{16504}{400709367}\;\frac{17}{1053}$ $\frac{2784}{16504}\;\frac{0}{0}$
5	$\frac{12}{56}\;\frac{1}{6}$ $\frac{12}{12}\;\frac{0}{0}$	$\frac{113}{7882}\;\frac{3}{34}$ $\frac{106}{113}\;\frac{0}{0}$	$\frac{967}{1757384}\;\frac{6}{190}$ $\frac{568}{967}\;\frac{0}{0}$	$\frac{7860}{400709367}\;\frac{11}{1053}$ $\frac{2784}{7860}\;\frac{0}{0}$	$\frac{61245}{79846389608}\;\frac{18}{5624}$ $\frac{13436}{61245}\;\frac{42}{5624}$
6	$\frac{14}{84}\;\frac{1}{7}$ $\frac{14}{14}\;\frac{0}{0}$	$\frac{160}{27412}\;\frac{3}{50}$ $\frac{152}{160}\;\frac{0}{0}$	$\frac{1746}{20075154}\;\frac{6}{386}$ $\frac{1076}{1746}\;\frac{0}{0}$	$\frac{19123}{17315935276}\;\frac{11}{3250}$ $\frac{7926}{19123}\;\frac{0}{0}$	$\frac{211177}{13581262890860}\;\frac{18}{28576}$ $\frac{63762}{211177}\;\frac{0}{0}$
7	$\frac{16}{120}\;\frac{1}{8}$ $\frac{16}{16}\;\frac{0}{0}$	$\frac{217}{85822}\;\frac{3}{70}$ $\frac{208}{217}\;\frac{0}{0}$	$\frac{2985}{200210860}\;\frac{6}{734}$ $\frac{1932}{2985}\;\frac{0}{0}$	$\frac{44232}{646805806837}\;\frac{11}{9343}$ $\frac{21506}{44232}\;\frac{0}{0}$	$\frac{709894}{1994012193306252}\;\frac{18}{136758}$ $\frac{292654}{709894}\;\frac{0}{0}$
8	$\frac{18}{165}\;\frac{1}{9}$ $\frac{18}{18}\;\frac{0}{0}$	$\frac{286}{246202}\;\frac{3}{95}$ $\frac{276}{286}\;\frac{0}{0}$	$\frac{4906}{1774852035}\;\frac{6}{1324}$ $\frac{3324}{4906}\;\frac{0}{0}$	$\frac{99058}{21250295114566}\;\frac{11}{25207}$ $\frac{55714}{99058}\;\frac{0}{0}$	$\frac{2389830}{256826902064216489}\;\frac{18}{613894}$ $\frac{1280816}{2389830}\;\frac{0}{0}$

TABLE 2. Enumeration of Precodes

6. CATEGORICAL VIEW OF PRECODES

Recall the preliminary definitions from Appendix B. Throughout this section, we use $\mathfrak{P}$ and $\mathfrak{C}$ to denote the categories of precodes and codes, respectively.

6.1. Well-powered and Co-(well-powered). Recall Definition B.13.

Theorem 6.1. *The categories $\mathfrak{P}$ and $\mathfrak{C}$ are well-powered and co-(well-powered).*

Proof. Let $\mathcal{B}$ be a precode. By definition, each of $P_{\mathcal{B}}$, $C_{\mathcal{B}}$, $e_{\mathcal{B}}$, and $d_{\mathcal{B}}$ is a set. Since we are only concerned with considering subobjects up to isomorphism, we may (by relabeling) restrict our attention to subprecodes of $\mathcal{B}$. That is, we need only consider subobjects of $\mathcal{B}$ of the form $(\mathcal{A}, (1_{P_{\mathcal{A}}}, 1_{C_{\mathcal{A}}}))$, so that $P_{\mathcal{A}} \subseteq P_{\mathcal{B}}$, $C_{\mathcal{A}} \subseteq C_{\mathcal{B}}$, $e_{\mathcal{A}} \subseteq e_{\mathcal{B}}$, and $d_{\mathcal{A}} \subseteq d_{\mathcal{B}}$. Each subprecode of $\mathcal{B}$ is an element in $X = 2^{P_{\mathcal{A}}} \times 2^{C_{\mathcal{B}}} \times 2^{e_{\mathcal{B}}} \times 2^{d_{\mathcal{B}}}$. Since the power set of a set is again a set, and since the product of two sets is a set, then X is a set. Thus, there is at most a set of subprecodes of $\mathcal{B}$, and $\mathfrak{P}$ is well-powered.

Now, let $(f = (f_1, f_2), A)$ be a quotient object of $\mathcal{B}$. We can view the quotient object $(f, \mathcal{A})$ as the 3-tuple $(f_1, f_2, \mathcal{A})$. Since f_1 and f_2 are surjective and since we are only concerned with considering quotient objects up to isomorphism, then by relabeling, we may assume that $P_{\mathcal{A}} \subseteq P_{\mathcal{B}}$, $C_{\mathcal{A}} \subseteq C_{\mathcal{B}}$, $e_{\mathcal{A}} \subseteq e_{\mathcal{B}}$, and $d_{\mathcal{A}} \subseteq d_{\mathcal{B}}$. As above, there are at most a set of such precodes $\mathcal{A}$.

We further note that $(f_1, P_{\mathcal{A}})$ and $(f_2, C_{\mathcal{A}})$ are quotient objects of $P_{\mathcal{B}}$ and $C_{\mathcal{B}}$, respectively, in the category of sets. As in [6], the category of sets is co-(well-powered). Thus, for each $\mathcal{A}$, the class of possible functions for f_1 is at most a set, and similarly for f_2. Thus, there is at most a set of 3-tuples of the form $(f_1, f_2, \mathcal{A})$. Hence, $\mathfrak{P}$ is co-(well-powered).

Since $\mathfrak{C}$ is a subcategory of $\mathfrak{P}$, then $\mathfrak{C}$ is well-powered and co-(well-powered). $\square$

6.2. Intersections. Recall Definition B.14.

Theorem 6.2. *The categories $\mathfrak{P}$ and $\mathfrak{C}$ have intersections. We note that the following theorem holds since the categories have limits as shown in [4]. However, the following alternate proof is instructive since it gives the construction of an intersection within the categories.*

Proof. Let $(\mathcal{A}_i = (P_i, C_i, e_i, d_i), m_i)$ be a family of subobjects of a precode $\mathcal{A}$ indexed by a set I. We will construct an intersection of the family $(\mathcal{A}_i, m_i)$ in $\mathfrak{P}$. W.l.o.g., we may assume that for each $i \in I$, $\mathcal{A}_i$ is a subprecode of $\mathcal{A}$; that is, $m_i = 1_{\mathcal{A}_i} = (1_{P_i}, 1_{C_i})$.

Let $\bigcap_{i \in I} \mathcal{A}_i$ denote the precode $(\bigcap_{i \in I} P_i, \bigcap_{i \in I} C_i, \bigcap_{i \in I} e_i, \bigcap_{i \in I} d_i)$. Let

$$1_{P_{\mathcal{A}}} : \bigcap_{i \in I} P_i \longrightarrow P_{\mathcal{A}} \text{ and } 1_{C_{\mathcal{A}}} : \bigcap_{i \in I} C_i \longrightarrow C_{\mathcal{A}}$$

be the usual inclusions. We claim that $(\bigcap_{i \in I} \mathcal{A}_i, 1_{\mathcal{A}} = (1_{P_{\mathcal{A}}}, 1_{C_{\mathcal{A}}}))$ is an intersection of the family $(\mathcal{A}_i, m_i)$.

It is clear that (1) and (2) in Defintion B.14 hold. Now, suppose $g : \mathcal{B} \longrightarrow \mathcal{A}$ and $g_i : \mathcal{B} \longrightarrow \mathcal{A}_i$ (for each $i \in I$) are homomorphisms such that $g = m_i \circ g_i = 1_{\mathcal{A}_i} \circ g_i = g_i$. We need to show that there exists a unique homomorphism $f : \mathcal{B} \longrightarrow \bigcap_{i \in I} \mathcal{A}_i$ such that $1_{\mathcal{A}} \circ f = g$. But, this is clear since we must have $f = g$.

Since the intersection of a family of codes is again a code, then $\mathfrak{C}$ has intersections as well. □

6.3. Pullbacks and Pushouts. Recall Definition B.45.

Lemma 6.3. *In the categories $\mathfrak{P}$ and $\mathfrak{C}$, the pullback of*

(1) an epimorphism is an epimorphism.

(2) a regular epimorphism is a regular epimorphism.

(2) a monomorphism is a monomorphism.

(3) a regular monomorphism is a regular monomorphism.

(4) a retraction is a retraction.

Proof. Statments (1) and (2) are proven in Lemmas 19 and 20 in [4]. The remainder of the lemma holds by Proposition 21.13 in [6]. □

6.4. **Regular Monomorphisms.** Recall Definition B.22.

Theorem 6.4. *Let $h = (h_1, h_2) : \mathcal{H} \longrightarrow \mathcal{A}$ be a precode monomorphism. Then h is a regular monomorphism in $\mathfrak{P}$ if and only if*

$$h : \mathcal{H} \longrightarrow \mathcal{F} = (h_1(P_{\mathcal{H}}), h_2(C_{\mathcal{H}}), e_{\mathcal{A}} \cap (h_1(P_{\mathcal{H}}) \times h_2(C_{\mathcal{H}})), d_{\mathcal{A}} \cap (h_2(C_{\mathcal{H}}) \times h_1(P_{\mathcal{H}})))$$

is a strong monomorphism; that is, if and only if it is an isomorphism.

Proof. By relabeling, we may suppose that $h = 1_{\mathcal{H}}$. We also suppose that for each $\pi \in P_{\mathcal{A}}$, $\pi' \notin P_{\mathcal{A}}$ and for each $\kappa \in C_{\mathcal{A}}$, $\kappa' \notin C_{\mathcal{A}}$.

Case 1: Suppose $P_{\mathcal{H}} = \emptyset$ or $C_{\mathcal{H}} = \emptyset$. Then $h_1(P_{\mathcal{H}}) = \emptyset$ or $h_2(C_{\mathcal{H}}) = \emptyset$, and h is a strong monomorphism onto $\mathcal{F} = (h_1(P_{\mathcal{H}}), h_2(C_{\mathcal{H}}), \emptyset, \emptyset)$. Thus, we need only show that h is regular.

If $P_{\mathcal{H}} = C_{\mathcal{H}} = \emptyset$, then h is the empty monomorphism and is regular since $h \approx Equ(h, h)$. So, suppose that exactly one of $P_{\mathcal{H}}$ and $C_{\mathcal{H}}$ is nonempty, say $P_{\mathcal{H}} \neq \emptyset$. Thus, $\mathcal{H} = (P_{\mathcal{H}}, \emptyset, \emptyset, \emptyset)$. Fix $\psi \in P_{\mathcal{H}}$. For each $\pi \in P_{\mathcal{A}}$, we define

$$g_1(\pi) = \begin{cases} \pi, & \text{if } \pi \in P_{\mathcal{H}} \\ \psi, & \text{if } \pi \notin P_{\mathcal{H}}. \end{cases}$$

Let $C'_{\mathcal{A}} = \{\kappa' \mid \kappa \in C_{\mathcal{A}}\}$ and $C = C_{\mathcal{A}} \cup C'_{\mathcal{A}}$. If $C_{\mathcal{A}} \neq \emptyset$, then for each $\kappa \in C_{\mathcal{A}}$, define $f_2(\kappa) = \kappa$ and $g_2(\kappa) = \kappa'$. If $C_{\mathcal{A}} = \emptyset$, we let $f_2 = g_2 : C_{\mathcal{A}} \longrightarrow C$ be the empty function. Let

$$e = e_{\mathcal{A}} \cup \{(\pi, \kappa') \mid (\pi, \kappa) \in e_{\mathcal{A}}\} \text{ and } d = d_{\mathcal{A}} \cup \{(\kappa', \pi) \mid (\kappa, \pi) \in d_{\mathcal{A}}\}.$$

Then $f = (1_{P_{\mathcal{A}}}, f_2)$ and $g = (g_1, g_2)$ are homomorphisms from $\mathcal{A}$ to $(P_{\mathcal{A}}, C, e, d)$. By Lemma B.21, $Equ(f, g) \approx (P_{\mathcal{H}}, \emptyset, \emptyset, \emptyset) = \mathcal{H}$. Thus, h is a regular monomorphism.

Case 2: Suppose that $P_{\mathcal{H}} \neq \emptyset$ and $C_{\mathcal{H}} \neq \emptyset$.

$(\Leftarrow)$: Suppose h is a strong monomorphism onto $\mathcal{F}$. Fix $\psi \in P_{\mathcal{H}}$ and $\chi \in C_{\mathcal{H}}$. Set $f = 1_{\mathcal{A}}$ and $g = (g_1, g_2)$, where for $\pi \in P_{\mathcal{A}}$ and $\kappa \in C_{\mathcal{A}}$, we define

$$g_1(\pi) = \begin{cases} \pi, & \text{if } \pi \in P_{\mathcal{H}} \\ \psi, & \text{if } \pi \notin P_{\mathcal{H}}, \end{cases} \quad \text{and} \quad g_2(\kappa) = \begin{cases} \kappa, & \text{if } \kappa \in C_{\mathcal{H}} \\ \chi, & \text{if } \kappa \notin C_{\mathcal{H}}. \end{cases}$$

Note that $f, g : \mathcal{A} \longrightarrow \mathcal{B} = (P_{\mathcal{A}}, C_{\mathcal{A}}, P_{\mathcal{A}} \times C_{\mathcal{A}}, C_{\mathcal{A}} \times P_{\mathcal{A}})$ are homomorphisms since their components are functions. Since h is strong onto $\mathcal{F}$, then $e_{\mathcal{A}}|_{P_{\mathcal{H}} \times C_{\mathcal{H}}} = e_{\mathcal{H}}$ and $d_{\mathcal{A}}|_{C_{\mathcal{H}} \times P_{\mathcal{H}}} = d_{\mathcal{H}}$. By Lemma B.21, then $Equ(f, g) \approx \mathcal{H}$, and h is a regular monomorphism.

$(\Rightarrow)$: Suppose h is a regular monomorphism. Thus, $\mathcal{H}$ is the equalizer of some precode homomorphisms $f, g : \mathcal{A} \longrightarrow \mathcal{B}$. By Lemma B.21, $e_{\mathcal{H}} = e_{\mathcal{A}}|_{P_{\mathcal{H}} \times C_{\mathcal{H}}}$ and $d_{\mathcal{H}} = d_{\mathcal{A}}|_{C_{\mathcal{H}} \times P_{\mathcal{H}}}$. Thus, h is a strong homomorphism onto $\mathcal{F}$. $\square$

Theorem 6.5. *Let $h = (h_1, h_2) : \mathcal{H} \longrightarrow \mathcal{A}$ be a monomorphism between codes. Then h is a regular monomorphism in $\mathfrak{C}$ if and only if it is a strong monomorphism onto $\mathcal{F} = (h_1(P_{\mathcal{H}}), h_2(C_{\mathcal{H}}), e_{\mathcal{A}} \cap (h_1(P_{\mathcal{H}}) \times h_2(C_{\mathcal{H}})), d_{\mathcal{A}} \cap (h_2(C_{\mathcal{H}}) \times h_1(P_{\mathcal{H}})))$ and there is no $(\pi, \kappa) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$ such that $\pi \in P_{\mathcal{A}} \backslash h_1(P_{\mathcal{H}})$ and $\kappa \in h_2(C_{\mathcal{H}})$.*

Proof. If $P_{\mathcal{H}} = \emptyset$ or $C_{\mathcal{H}} = \emptyset$, then the proof is the same as in Case 1 of Theorem 6.4. Thus, we suppose that $P_{\mathcal{H}} \neq \emptyset$ and $C_{\mathcal{H}} \neq \emptyset$. By relabeling, we may suppose that $h = 1_{\mathcal{H}}$. We also suppose that for each $\pi \in P_{\mathcal{A}}$, $\pi' \notin P_{\mathcal{A}}$ and for each $\kappa \in C_{\mathcal{A}}$, $\kappa' \notin C_{\mathcal{A}}$.

$(\Leftarrow)$: Suppose h is a strong monomorphism onto $\mathcal{F}$ and there is no $(\pi, \kappa) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$ such that $\pi \in P_{\mathcal{A}} \backslash P_{\mathcal{H}}$ and $\kappa \in C_{\mathcal{H}}$. Let $P = P_{\mathcal{A}} \backslash P_{\mathcal{H}}$ and $C = C_{\mathcal{A}} \backslash C_{\mathcal{H}}$. Let

$$e = ((P \times C_{\mathcal{H}}) \cup (P_{\mathcal{H}} \times C) \cup (P \times C)) \cap e_{\mathcal{A}};$$

that is, e contains all the edges in $e_{\mathcal{A}}$ incident on at least one vertex not in $\mathcal{H}$. Similarly, let $d = ((C_{\mathcal{H}} \times P) \cup (C \times P_{\mathcal{H}}) \cup (C \times P)) \cap d_{\mathcal{A}}$.

Let $P' = \{\pi' \mid \pi \in P\}$, $C' = \{\kappa' \mid \kappa \in C\}$,

$e' = \{(\pi', \kappa) \mid (\pi, \kappa) \in e\} \cup \{(\pi, \kappa') \mid (\pi, \kappa) \in e\}$ and $d' = \{(\kappa, \pi') \mid (\kappa, \pi) \in d\} \cup \{(\kappa', \pi) \mid (\kappa, \pi) \in d\}$.

Let $\mathcal{B} = (P_\mathcal{B}, C_\mathcal{B}, e_\mathcal{B}, d_\mathcal{B}) = (P_\mathcal{A} \cup P', C_\mathcal{A} \cup C', e_\mathcal{A} \cup e', d_\mathcal{A} \cup d')$. We now show that $\mathcal{B}$ is a code by

contradiction. Assume there are $\pi_1, \pi_2 \in P_\mathcal{B}$ with $\pi_1 \neq \pi_2$ and $\chi \in C_\mathcal{B}$ such that $(\pi_1, \chi) \in e_\mathcal{B}$ and

$(\chi, \pi_2) \in d_\mathcal{B}$. Since $C_\mathcal{B} = C_\mathcal{A} \cup C'$, we have two cases.

Case 1: Suppose that $\chi \in C_\mathcal{A}$. Since $\mathcal{A}$ is a code, then either $\pi_1 \notin P_\mathcal{A}$ or $\pi_2 \notin P_\mathcal{A}$. That is,

$\pi_1 \in P'$ or $\pi_2 \in P'$. W.l.o.g., suppose $\pi_1 \in P'$. Then $\pi_1 = \pi'$ for some $\pi \in P$, and $(\pi, \chi) \in e \subseteq e_\mathcal{A}$

since $(\pi', \chi) \in e'$. Since $(\chi, \pi_2) \in d_\mathcal{B} = d_\mathcal{A} \cup d'$, then either $(\chi, \pi_2) \in d_\mathcal{A}$ or $(\chi, \pi_2) \in d'$. If

$(\chi, \pi_2) \in d_\mathcal{A}$, then $\pi_2 = \pi$ since $\mathcal{A}$ is a code. But, then $(\pi, \chi) \in e_\mathcal{A} \cap d_\mathcal{A}^{nv}$, a contradiction. Thus,

$(\chi, \pi_2) \in d'$. Since $\chi \in C_\mathcal{A}$, then $\pi_2 = \psi'$ for some $\psi \in P_\mathcal{A}$ and $(\chi, \psi) \in d \subseteq d_\mathcal{A}$. Since, $(\pi, \chi) \in e_\mathcal{A}$

and $(\chi, \psi) \in d_\mathcal{A}$, then $\pi = \psi$. Hence, $\pi_1 = \pi' = \psi' = \pi_2$, a contradiction.

Case 2: Suppose that $\chi \in C'$. Then $\chi = \kappa'$ for some $\kappa \in C$. Thus, $(\pi_1, \kappa) \in e_\mathcal{A}$ and $(\kappa, \pi_2) \in d_\mathcal{A}$.

Since $\mathcal{A}$ is a code, then $\pi_1 = \pi_2$, a contradiction.

Thus, $\mathcal{B}$ is a code. We now construct $g = (g_1, g_2) : A \longrightarrow B$ so that $Equ(1_\mathcal{A}, g) \approx \mathcal{H}$. For

$\pi \in P_\mathcal{A}$ and $\kappa \in C_\mathcal{A}$, we define

$$g_1(\pi) = \begin{cases} \pi, & \text{if } \pi \in P_\mathcal{H} \\ \pi', & \text{if } \pi \notin P_\mathcal{H}, \end{cases} \quad \text{and} \quad g_2(\kappa) = \begin{cases} \kappa, & \text{if } \kappa \in C_\mathcal{H} \\ \kappa', & \text{if } \kappa \notin C_\mathcal{H}. \end{cases}$$

It is clear from the definition of $\mathcal{B}$ that g is a homomorphism. Since h is a strong monomorphism

onto $\mathcal{F}$, then $e_\mathcal{A}|_{P_\mathcal{H} \times C_\mathcal{H}} = e_\mathcal{H}$ and $d_\mathcal{A}|_{C_\mathcal{H} \times P_\mathcal{H}} = d_\mathcal{H}$. By Lemma B.21, $Equ(1_\mathcal{A}, g) \approx \mathcal{H}$. Thus, h is

a regular monomorphism.

$(\Rightarrow)$: Suppose h is a regular monomorphism. Hence, there is a code $\mathcal{B}$ such that $\mathcal{H}$ is the equalizer

of some homomorphisms $f, g : \mathcal{A} \longrightarrow \mathcal{B}$. By Lemma B.21, $e_\mathcal{H} = e_\mathcal{A}|_{P_\mathcal{H} \times C_\mathcal{H}}$ and $d_\mathcal{H} = d_\mathcal{A}|_{C_\mathcal{H} \times P_\mathcal{H}}$.

Thus, h is a strong homomorphism onto $\mathcal{F}$.

Now, assume there are $\pi \in P_\mathcal{A} \backslash h_1(P_\mathcal{H}) = P_\mathcal{A} \backslash P_\mathcal{H}$ and $\kappa \in h_2(C_\mathcal{H}) = C_\mathcal{H}$ such that $(\pi, \kappa) \in e_\mathcal{A} \cap$

$d_\mathcal{A}^{nv}$. Since $\mathcal{H}$ is the equalizer of $f = (f_1, f_2)$ and $g = (g_1, g_2)$, then $f_1(\pi) \neq g_1(\pi)$ since $\pi \notin P_\mathcal{H}$, and

$f_2(\kappa) = g_2(\kappa)$ since $\kappa \in C_{\mathcal{H}}$. Since $(\pi, \kappa) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$, then $(f_1(\pi), f_2(\kappa)), (g_1(\pi), g_2(\kappa)) \in e_{\mathcal{B}} \cap d_{\mathcal{B}}^{nv}$. This contradicts that $\mathcal{B}$ is a code since $f_1(\pi) \neq g_1(\pi)$ and $f_2(\kappa) = g_2(\kappa)$. $\qquad\qquad\square$

6.5. **Regular Epimorphisms.** We note that the following was proven by Dr. Klappenecker independently of this work and is recorded as Proposition 16 in [4].

Theorem 6.6. *Let $h : \mathcal{B} \longrightarrow \mathcal{H}$ be a precode or code epimorphism. Then h is a regular epimorphism if and only if it is a strong epimorphism.*

Proof. The following proof also holds if $\mathcal{B}$ and $\mathcal{H}$ are assumed to be codes.

($\Leftarrow$) : Suppose h is a strong epimorphism. Let (s, t) be the kernel of h. We recall that $\mathcal{H}$ is isomorphic to $\mathcal{H}' = (P_{\mathcal{B}}/s, C_{\mathcal{B}}/t, e_{\mathcal{B}}/(s, t), d_{\mathcal{B}}/(t, s))$.

Let $\{P_i\}_{i \in I}$ be the family of s-equivalence classes of $P_{\mathcal{B}}$ and $\{C_j\}_{j \in J}$ be the family of t-equivalence classes of $C_{\mathcal{B}}$. Consider the code $\mathcal{A} = (P_{\mathcal{A}} = \bigcup_{i \in I}(P_i \times P_i), C_{\mathcal{A}} = \bigcup_{j \in J}(C_j \times C_j), \emptyset, \emptyset)$.

For $(\pi, \psi) \in P_{\mathcal{A}}$, we define $f_1((\pi, \psi)) = \pi$ and $g_1((\pi, \psi)) = \psi$. Similarly, for $(\kappa, \chi) \in C_{\mathcal{A}}$, we define $f_2((\kappa, \chi)) = \kappa$ and $g_2((\kappa, \chi)) = \chi$. Since $e_{\mathcal{A}} = \emptyset = d_{\mathcal{A}}$, then $f = (f_1, f_2), g = (g_1, g_2) : \mathcal{A} \longrightarrow B$ are trivially precode homomorphisms.

Let E_1 be the smallest equivalence relation on $P_{\mathcal{B}}$ containing the pairs $(f_1((\pi, \psi)), g_1((\pi, \psi))) = (\pi, \psi)$ for all $(\pi, \psi) \in P_{\mathcal{A}}$, and let E_2 be the smallest equivalence relation on $C_{\mathcal{B}}$ containing the pairs $(f_2((\kappa, \chi)), g_2((\kappa, \chi))) = (\kappa, \chi)$ for all $(\kappa, \chi) \in C_{\mathcal{A}}$. It is clear that $E_1 = s$ and $E_2 = t$. As in [4], $Coeq(f, g) \approx (P_{\mathcal{B}}/E_1, C_{\mathcal{B}}/E_2, e_{\mathcal{B}}/(E_1, E_2), d_{\mathcal{B}}/(E_2, E_1)) = \mathcal{H}'$, which is isomorphic to $\mathcal{H}$. So, h is a regular epimorphism.

($\Rightarrow$) : Suppose h is a regular epimorphism. Thus, h is the coequalizer of precode homomorphisms $f, g : A \longrightarrow B$. Assume that h is not strong so that $h_2 \circ e_{\mathcal{B}} \circ h_1^{nv} \neq e_{\mathcal{H}}$ or $h_1 \circ d_{\mathcal{B}} \circ h_2^{nv} \neq d_{\mathcal{H}}$. W.l.o.g., suppose that $h_2 \circ e_{\mathcal{B}} \circ h_1^{nv} \neq e_{\mathcal{H}}$. Hence, there exist $\pi \in P_{\mathcal{H}}$ and $\kappa \in C_{\mathcal{H}}$ such that $(\pi, \kappa) \in e_{\mathcal{H}}$, but there exist no $\pi' \in P_{\mathcal{B}}$ and $\kappa' \in C_{\mathcal{B}}$ for which $h_1(\pi') = \pi$, $h_2(\kappa') = \kappa$, and $(\pi', \kappa') \in e_{\mathcal{B}}$. Letting $\mathcal{H}' = (P_{\mathcal{H}}, C_{\mathcal{H}}, e_{\mathcal{H}} \backslash \{(\pi, \kappa)\}, d_{\mathcal{H}})$, then $h : B \longrightarrow \mathcal{H}'$ satisfies $h \circ f = h \circ g$.

However, $(1_{P_{\mathcal{H}}}, 1_{C_{\mathcal{H}}}) : \mathcal{H} \longrightarrow \mathcal{H}'$ is not a homomorphism, contradicting that $(h, \mathcal{H}) \approx Coeq(f, g)$.

Thus, h must be a strong homomorphism. $\qquad\qquad\square$

6.6. Extremal Monomorphisms. Recall Definition B.23.

Theorem 6.7. *In $\mathfrak{P}$ and $\mathfrak{C}$, a monomorphism*

$$h = (h_1, h_2) : \mathcal{H} \longrightarrow \mathcal{A}$$

is an extremal monomorphism if and only if

$$h : \mathcal{H} \longrightarrow \mathcal{F} = (h_1(P_{\mathcal{H}}), h_2(C_{\mathcal{H}}), e_{\mathcal{A}} \cap (h_1(P_{\mathcal{H}}) \times h_2(C_{\mathcal{H}})), d_{\mathcal{A}} \cap (h_2(C_{\mathcal{H}}) \times h_1(P_{\mathcal{H}})))$$

is a strong monomorphism; that is, if and only if it is an isomorphism.

Proof. By relabeling, we suppose that $h = 1_{\mathcal{H}}$.

$(\Rightarrow)$: Suppose that h is not an isomorphism onto $\mathcal{F} = (P_{\mathcal{H}}, C_{\mathcal{H}}, e_{\mathcal{A}} \cap (P_{\mathcal{H}} \times C_{\mathcal{H}}), d_{\mathcal{A}} \cap (C_{\mathcal{H}} \times P_{\mathcal{H}}))$.
Let $f = 1_{\mathcal{H}} : \mathcal{F} \longrightarrow \mathcal{F}$. Since $h = f \circ h$ and $h : \mathcal{H} \longrightarrow \mathcal{F}$ is an epimorphism which is not an isomorphism, then $h : \mathcal{H} \longrightarrow \mathcal{A}$ is not extremal.

$(\Leftarrow)$: Suppose that h is an isomorphism onto $\mathcal{F} = (P_{\mathcal{H}}, C_{\mathcal{H}}, e_{\mathcal{A}} \cap (P_{\mathcal{H}} \times C_{\mathcal{H}}), d_{\mathcal{A}} \cap (C_{\mathcal{H}} \times P_{\mathcal{H}}))$.
Suppose that $h = f \circ e$ for some epimorphism $e = (e_1, e_2) : \mathcal{H} \longrightarrow \mathcal{B}$ and some homomorphism $f = (f_1, f_2) : \mathcal{B} \longrightarrow \mathcal{A}$. To show that h is extremal, we must show that e must be an isomorphism. Since h is a monomorphism, then e must be a monomorphism. Thus, e_1 and e_2 are bijections. Since $h_1 : P_{\mathcal{H}} \longrightarrow P_{\mathcal{H}}$ and $h_2 : C_{\mathcal{H}} \longrightarrow C_{\mathcal{H}}$ are bijections satisfying $h_1 = f_1 \circ e_1$ and $h_2 = f_2 \circ e_2$, then $f_1 : P_{\mathcal{B}} \longrightarrow P_{\mathcal{H}}$ and $f_2 : C_{\mathcal{B}} \longrightarrow C_{\mathcal{H}}$ must be bijections. Since $h : \mathcal{H} \longrightarrow \mathcal{F}$ is a strong epimorphism, then $f : \mathcal{B} \longrightarrow \mathcal{F}$ must be a strong epimorphism. Since strong precode bimorphisms are precode isomorphisms, then $f : \mathcal{B} \longrightarrow \mathcal{F}$ is a precode isomorphism. Since $h = f \circ e$, then e must be an isomorphism. $\qquad\square$

Corollary 6.8. *In $\mathfrak{P}$, the extremal monomorphisms are precisely the regular monomorphisms.*

Proof. This is just a combination of Theorems 6.4 and 6.7. $\qquad\square$

6.7. Extremal Epimorphisms. Recall Definition B.23.

Theorem 6.9. *In $\mathfrak{P}$ and $\mathfrak{C}$, the extremal epimorphisms are precisely the regular epimorphisms.*

Proof. This follows from Theorem 6.10 and Proposition B.27. □

6.8. Factorizations. We now prove the following theorem, noting that Dr. Klappenecker indpendently showed that the categories $\mathfrak{P}$ and $\mathfrak{C}$ are uniquely (regular epi, mono)-factorizable.

Theorem 6.10. *The categories $\mathfrak{P}$ and $\mathfrak{C}$ are uniquely (regular epi, mono)-factorizable and uniquely (extremal epi, mono)-factorizable.*

Proof. Let $f = (f_1, f_2) : \mathcal{A} \longrightarrow \mathcal{B}$ be a precode homomorphism. Recall from Definition A.19 that the image of f is the precode $Im(f) = (f_1(P_\mathcal{A}), f_2(C_\mathcal{A}), f_2 \circ e_\mathcal{A} \circ f_1^{nv}, f_1 \circ d_\mathcal{A} \circ f_2^{nv})$ and that $e = (f_1, f_2) : \mathcal{A} \longrightarrow Im(f)$ is a strong precode epimorphism. By Theorem 6.6, e is a regular epimorphism. Let $m = (1_{f_1(P_\mathcal{A})}, 1_{f_2(C_\mathcal{A})}) : Im(f) \longrightarrow B$. Then m is a monomorphism and $f = m \circ e$. Thus, $\mathfrak{P}$ is (regular epi, mono)-factorizable. By Proposition B.27, $\mathfrak{P}$ is uniquely (regular epi, mono)-factorizable. Notice that if $\mathcal{A}$ and $\mathcal{B}$ are codes, then so is $Im(f)$ since a subprecode of a code is again a code. The categories are uniquely (extremal epi, mono)-factorizable since regular epimorphisms are extremal epimorphisms. □

Corollary 6.11. *The categories $\mathfrak{P}$ and $\mathfrak{C}$ are (regular epi, mono) categories.*

Proof. This follows directly from Theorem 6.10 above and Proposition B.28. □

Theorem 6.12. *The category $\mathfrak{P}$ is an (epi, regular mono) category and $\mathfrak{C}$ is an (epi, extremal mono) category.*

Proof. By Theorems 6.1, 6.2, and B.20, the categories have intersections and equalizers and are well-powered. Thus, by Theorem B.29, they are (epi, extremal mono) categories. By Theorem 6.8, in $\mathfrak{P}$, the regular monomorphisms are the extremal monomorphisms. Hence, $\mathfrak{P}$ is an (epi, regular mono) category. □

Remark 6.13. *We now show how to construct the unique (epi, extremal mono)-factorization of a*

given precode homomorphism $h = (h_1, h_2) : \mathcal{H} \longrightarrow \mathcal{A}$ *guaranteed by Theorem 6.12. Let*

$$\mathcal{F} = (P_{\mathcal{F}} = h_1(P_{\mathcal{H}}), C_{\mathcal{F}} = h_2(C_{\mathcal{H}}), e_{\mathcal{F}} = e_{\mathcal{A}} \cap (h_1(P_{\mathcal{H}}) \times h_2(C_{\mathcal{H}})), d_{\mathcal{F}} = d_{\mathcal{A}} \cap (h_2(C_{\mathcal{H}}) \times h_1(P_{\mathcal{H}}))).$$

Then $m = (1_{P_{\mathcal{F}}}, 1_{C_{\mathcal{F}}}) : \mathcal{F} \longrightarrow \mathcal{F}$ *is an extremal monomorphism by Theorem 6.7;* $h : \mathcal{H} \longrightarrow \mathcal{F}$ *is an*

epimorphism; and $h = m \circ h$.

Corollary 6.14. *In* $\mathfrak{P}$ *and* $\mathfrak{C}$,

    (1) The composition of extremal monomorphisms is an extremal monomorphism.

    (2) The intersection of extremal subobjects is an extremal subobject.

    (3) The inverse image (pullback) of an extremal monomorphism is an extremal monomorphism.

    (4) The product of extremal monomorphisms is an extremal monomorphism.

Proof. This follows from Theorems 6.12, B.29, and B.30.   □

6.9. Sections and Retractions. Recall Definition B.8.

Proposition 6.15. *In* $\mathfrak{P}$ *and* $\mathfrak{C}$, *there are regular monomorphisms which are not sections.*

Proof. Let $\mathcal{H} = (\{\pi\}, \{\kappa\}, \{(\pi, \kappa)\}, \emptyset)$ and $\mathcal{A} = (\{\pi, \psi\}, \{\kappa, \chi\}, \{(\pi, \kappa)\}, \{(\chi, \psi)\})$. Let $f_1(\pi) = \pi$

and $f_2(\kappa) = \kappa$. Then $f = (f_1, f_2) : \mathcal{H} \longrightarrow \mathcal{A}$ is a regular monomorphism, but it is not a section. To

see this, note that there is no homomorphism from $\mathcal{A}$ to $\mathcal{H}$ since the decode relation of $\mathcal{A}$ is empty

while the one for $\mathcal{A}$ is not.   □

Lemma 6.16. *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be precodes. If* $A \xrightarrow{f} B$ *is a section (or a retraction), then* $P_{\mathcal{A}} = \emptyset \Leftrightarrow$

$P_{\mathcal{B}} = \emptyset$, $C_{\mathcal{A}} = \emptyset \Leftrightarrow C_{\mathcal{B}} = \emptyset$, $e_{\mathcal{A}} = \emptyset \Leftrightarrow e_{\mathcal{B}} = \emptyset$, *and* $d_{\mathcal{A}} = \emptyset \Leftrightarrow d_{\mathcal{B}} = \emptyset$. *Furthermore, if* f *is a*

section, then a vertex in $\mathcal{A}$ *is isolated if and only if its image under* f *is isolated in* $\mathcal{B}$.

Proof. ($\Leftarrow$): These hold since f is a homomorphism.

($\Rightarrow$): These hold since if f is a section or a retraction, then there exists a precode homomorphism $g : \mathcal{B} \longrightarrow \mathcal{A}$. $\square$

Lemma 6.17. *Let $\mathcal{A}$ and $\mathcal{B}$ be precodes. If $f = (f_1, f_2) : \mathcal{A} \longrightarrow \mathcal{B}$ is a section, then it must be an isomorphism onto*

$$\mathcal{F} = (f_1(P_\mathcal{A}), f_2(C_\mathcal{A}), e_\mathcal{B} \cap (f_1(P_\mathcal{A}) \times f_2(C_\mathcal{A})), d_\mathcal{B} \cap (f_2(C_\mathcal{A}) \times f_1(P_\mathcal{A}))).$$

If f is a retraction, then it must be a strong epimorphism.

Proof. By Proposition 16.15 in [6], sections are regular monomorphisms, and dually, retractions are regular epimorphisms. The lemma then holds by Theorems 6.4 and 6.6. $\square$

Lemma 6.18. *Let $\mathcal{A}$ and $\mathcal{B}$ be precodes. Recall that we can view $\mathcal{A}$ and $\mathcal{B}$ as bipartite digraphs. If $\mathcal{A} \xrightarrow{f} \mathcal{B}$ is a section, then f must send distinct components of $\mathcal{A}$ into distinct components of $\mathcal{B}$.*

Proof. Recall that a precode homomorphism is a graph homomorphism between the associated digraphs. Thus, the image of a connected precode must be a connected precode. Now, if f is a section, then there exists a precode homomorphism $g : \mathcal{B} \longrightarrow \mathcal{A}$ such that $g \circ f = 1_\mathcal{A}$. Since g and f must each send components into components, the proof is complete. $\square$

Lemma 6.19. *Let $\mathcal{A}$ and $\mathcal{B}$ be precodes. Recall that we can view $\mathcal{A}$ and $\mathcal{B}$ as bipartite digraphs. If $\mathcal{B} \xrightarrow{g} \mathcal{A}$ is a retraction with associated section $\mathcal{A} \xrightarrow{f} \mathcal{B}$ (i.e. $g \circ f = 1_\mathcal{A}$) and $X_\mathcal{A}$ is a graph component of $\mathcal{A}$, then there must be a component $X_\mathcal{B}$ of $\mathcal{B}$ for which $X_\mathcal{A} = Im(g|_{X_\mathcal{B}})$. That is, $X_\mathcal{A}$ is the image of $X_\mathcal{B}$ under g. Furthermore, $f(X_\mathcal{A})$ is isomorphic to $X_\mathcal{A}$ and $g|_{f(X_\mathcal{A})} : f(X_\mathcal{A}) \longrightarrow X_\mathcal{A}$ is an isomorphism.*

Proof. Let $X_\mathcal{A}$ be a connected component of $\mathcal{A}$. Then $f(X_\mathcal{A})$ is a subprecode of some connected component $X_\mathcal{B}$ of $\mathcal{B}$. Since g is an epimorphism which must send components into components and since $g \circ f = 1_\mathcal{A}$, then $g(X_\mathcal{B}) = X_\mathcal{A}$. Since f is monic, then $f(X_\mathcal{A})$ is isomorphic to $X_\mathcal{A}$. Since $g \circ f = 1_\mathcal{A}$, then $g|_{f(X_\mathcal{A})} : f(X_\mathcal{A}) \longrightarrow X_\mathcal{A}$ is an isomorphism. $\square$

Lemma 6.20. *A monomorphism $f = (f_1, f_2) : \mathcal{A} = (P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A} = \emptyset, d_\mathcal{A} = \emptyset) \longrightarrow \mathcal{B}$ is a section if and only if the following conditions hold:*

(1) $\mathcal{B} = (P_\mathcal{B}, C_\mathcal{B}, e_\mathcal{B} = \emptyset, d_\mathcal{B} = \emptyset)$,

(2) $P_\mathcal{A} \neq \emptyset \Leftrightarrow P_\mathcal{B} \neq \emptyset$, and

(3) $C_\mathcal{A} \neq \emptyset \Leftrightarrow C_\mathcal{B} \neq \emptyset$.

Proof. ($\Rightarrow$): This holds by Lemma 6.16.

($\Leftarrow$): As mentioned on page 33 in [6], a morphism in the category of sets is a section if and only if it is injective and is not the empty function from the empty set to a non-empty set. Thus, f_1 and f_2 are sections in the category of sets. Since the encode and decode relations of $\mathcal{A}$ and $\mathcal{B}$ are empty, f is clearly a section in $\mathfrak{P}$ and $\mathfrak{C}$. $\square$

Lemma 6.21. *Let $\mathcal{A}$ be a self-companion code with $e_\mathcal{A} \neq \emptyset$ or $d_\mathcal{A} \neq \emptyset$. Then a precode monomorphism $f : \mathcal{A} \longrightarrow \mathcal{B}$ is a section if and only if it is an isomorphism onto $\mathcal{F} = (f_1(P_\mathcal{A}), f_2(C_\mathcal{A}), e_\mathcal{B} \cap (f_1(P_\mathcal{A}) \times f_2(C_\mathcal{A})), d_\mathcal{B} \cap (f_2(C_\mathcal{A}) \times f_1(P_\mathcal{A})))$ which sends distinct components into distinct components and which sends isolated vertices to isolated vertices.*

Proof. ($\Rightarrow$): This holds by Lemmas 6.16, 6.17, and 6.18.

($\Leftarrow$): Suppose f is an isomorphism onto $\mathcal{F}$ which sends distinct components of $\mathcal{A}$ into distinct components of $\mathcal{B}$ and sends isolated vertices to isolated vertices. W.l.o.g., we suppose that $f = (1_{P_\mathcal{A}}, 1_{C_\mathcal{A}})$. Since $\mathcal{A}$ is self-companion, then $e_\mathcal{A} \neq \emptyset$ or $d_\mathcal{A} \neq \emptyset$ implies that there are $\psi \in P_\mathcal{A}$ and $\chi \in C_\mathcal{A}$ such that $(\psi, \chi) \in e_\mathcal{A} \cap d_\mathcal{A}^{nv}$. We define $g = (g_1, g_2) : \mathcal{B} \longrightarrow \mathcal{A}$ by showing how it behaves on a connected component $X_\mathcal{B} = (P_{X_\mathcal{B}}, C_{X_\mathcal{B}}, e_{X_\mathcal{B}}, d_{X_\mathcal{B}})$ in the digraph representation of $\mathcal{B}$.

By Lemma 6.18, either $X_\mathcal{B} \cap Im(f) = \emptyset$ or there is a component $X_\mathcal{A} = (P_{X_\mathcal{A}}, C_{X_\mathcal{A}}, e_{X_\mathcal{A}}, d_{X_\mathcal{A}})$ of $\mathcal{A}$ such that $X_\mathcal{B} \cap Im(f) = X_\mathcal{A}$. In the first case, for all $\pi \in P_{X_\mathcal{B}}$ and $\kappa \in C_{X_\mathcal{B}}$, we define $g_1(\pi) = \psi$ and $g_2(\kappa) = \chi$. In the latter case, if $X_\mathcal{B}$ is an isolated vertex, then we define g to be the identity on $X_\mathcal{B}$. If $X_\mathcal{B}$ is not an isolated vertex, then neither is $X_\mathcal{A}$. Since $\mathcal{A}$ is a self-companion

code, then $P_{X_A} = \{\psi'\}$ for some $\psi' \in P_A$, $C_{X_A} \neq \emptyset$, and $(\psi', \kappa') \in e_A \cap d_A^{nv}$ for each $\kappa' \in C_{X_A}$.

Fix some $\chi' \in C_{X_A}$. For each $\pi' \in P_{X_B}$, we define $g_1(\pi') = \psi'$. For each $\kappa \in C_{X_B}$, we define

$$g_2(\kappa) = \begin{cases} \kappa, & \text{if } \kappa \in C_{X_A} \\[2ex] \chi', & \text{if } \kappa \notin C_{X_A}. \end{cases}$$

It is clear that g is a homomorphism and that $g \circ f = 1_A$. Thus, f is a section. $\qquad\square$

We now give examples to show that the hypotheses of the above lemma cannot be weakened.

Example 6.22. *Let* $\mathcal{A} = (P_A = \{p_1\}, C_A = \{c_1, c_2\}, e_A = \{(p_1, c_1)\}, d_A = e_A^{nv})$ *and*

$$\mathcal{B} = (P_A, C_A, e_B = \{(p_1, c_1), (p_2, c_2)\}, d_A = e_B^{nv})$$

as depicted in Figure 3.
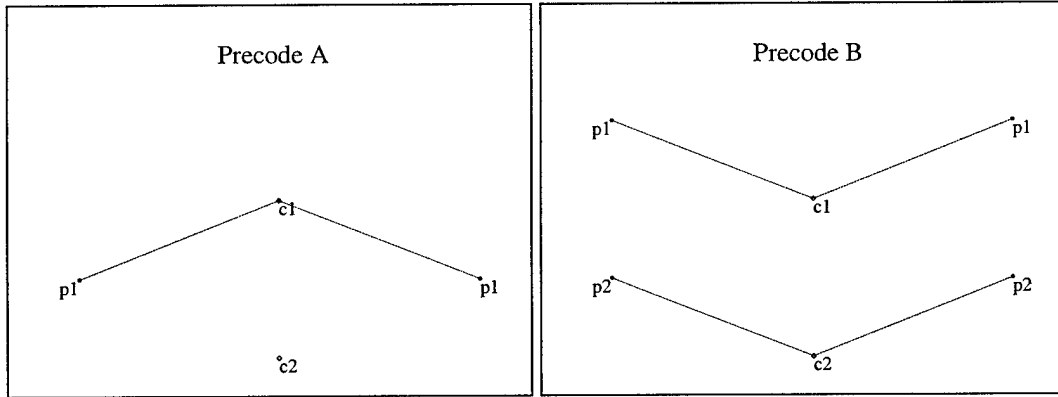
FIGURE 3. An Isolated Vertex Sent to a Nonisolated Vertex

Notice that $\mathcal{A}$ is a self-companion code and that $f = 1_A$ sends distinct components into distinct components. However, f sends an isolated vertex to a vertex which is not isolated. Since there is no homomorphism $g : \mathcal{B} \longrightarrow \mathcal{A}$ satisfying $g \circ f = 1_A$, then f is not a section.

Example 6.23. *Let* $\mathcal{A} = (P_{\mathcal{A}} = \{p_1, p_3\}, C_{\mathcal{A}} = \{c_1, c_2\}, e_{\mathcal{A}} = \{(p_1, c_1), (p_3, c_2)\}, e_{\mathcal{A}}^{nv})$ *and*

$$\mathcal{B} = (\{p_1, p_2, p_3\}, \{c_1, c_2\}, e_{\mathcal{B}} = \{(p_1, c_1), (p_2, c_2), (p_3, c_2)\}, d_{\mathcal{B}} = \{(c_1, p_1), (c_1, p_2), (c_2, p_3)\})$$

as depicted in Figure 4.
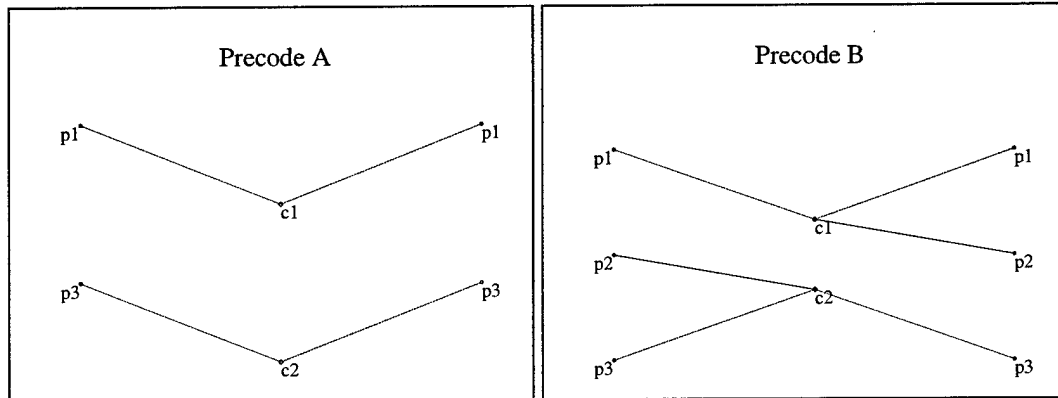
FIGURE 4. Distinct Components Not Sent to Distinct Components (Strip Chart Representation)

Notice that $\mathcal{A}$ *is a self-companion code with no isolated vertices. However,* $f = 1_{\mathcal{A}}$ *does not send distinct components into distinct components since* $\mathcal{B}$ *has only one component* **as a bipartite graph** *(see Figure 5).*
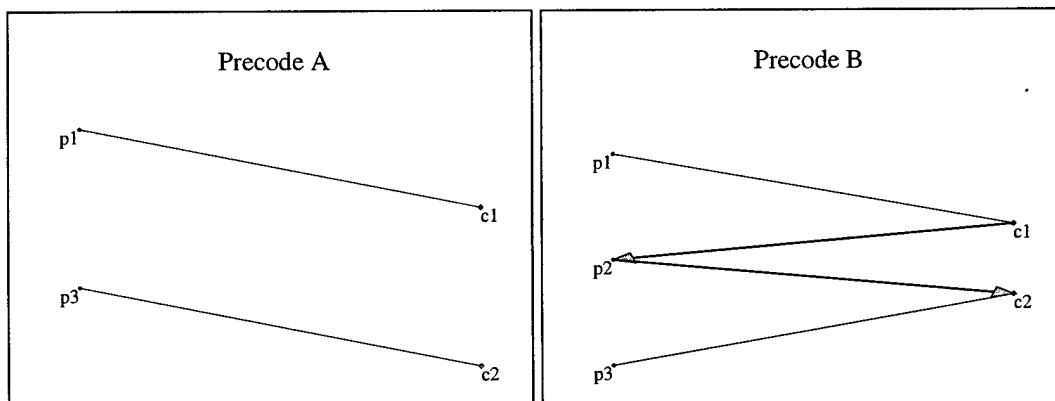
FIGURE 5. Distinct Components Not Sent to Distinct Components (Digraph Representation)

If $f = 1_A$ is a section, there must be a homomorphism $g = (g_1, g_2) : B \longrightarrow A$ such that $g_2 = 1_{C_A}$ and $g_1|_{P_A} = 1_{P_A}$. Also, $g_1(p_2)$ must be adjacent to both c_1 and c_2 in A since p_2 is adjacent to both c_1 and c_2 in B. Thus, f is not a section.

Example 6.24. *Let*

$$A = (P_A = \{p_1, p_2\}, C_A = \{c_1, c_2, c_3\}, e_A = \{(p_1, c_1), (p_1, c_2), (p_2, c_2), (p_2, c_3)\}, e_A^{nv})$$

and

$$B = (\{p_1, p_2, p_3\}, \{c_1, c_2, c_3\}, e_B = \{(p_1, c_1), (p_1, c_2), (p_2, c_2), (p_2, c_3), (p_3, c_1), (p_3, c_3)\}, e_B^{nv})$$

as depicted in Figure 6.
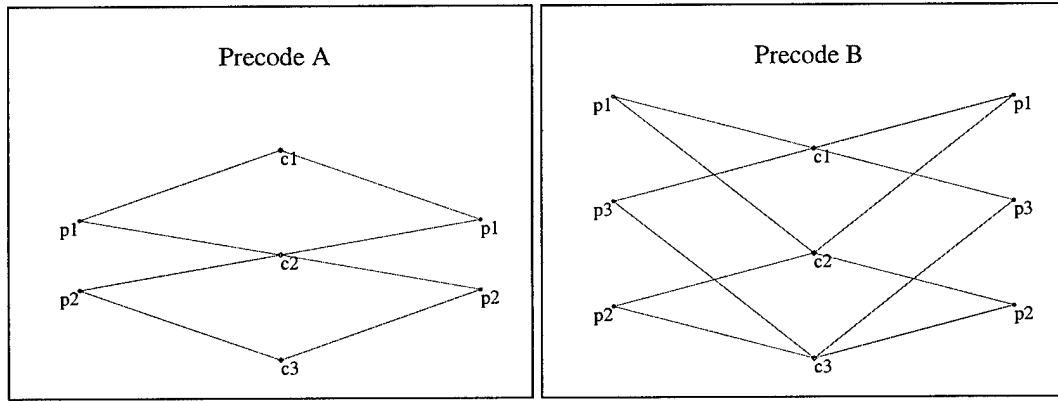
FIGURE 6. The Domain Is Not a Code

Notice that A is a self-companion precode with no isolated vertices and that $f = 1_A$ sends distinct components into distinct components. However, A is not a code. If f is a section, there must be a homomorphism $g = (g_1, g_2) : B \longrightarrow A$ such that $g_2 = 1_{C_A}$ and $g_1|_{P_A} = 1_{P_A}$. But, $g_1(p_3) \neq p_1$ since $(p_3, c_3) \in e_B$, but $(p_1, c_3) \notin e_A$. Similarly, $g_1(p_3) \neq p_2$. Thus, f is not a section.

Theorem 6.25. *Let $\mathcal{B}$ be a self-companion code, and let $\mathcal{A}$ be a precode. Then an epimorphism $\mathcal{B}\xrightarrow{g}\mathcal{A}$ is a retraction if and only if for each component $X_\mathcal{A}$ of $\mathcal{A}$, there is a component $X_\mathcal{B}$ of $\mathcal{B}$ for which $X_\mathcal{A} = g(X_\mathcal{B})$. We note that in this case, $\mathcal{A}$ will necessarily be a self-companion code as well.*

Proof. ($\Rightarrow$): This holds by Lemma 6.19.

($\Leftarrow$): Suppose the hypothesis holds. Then g is a strong epimorphism and $\mathcal{A}$ must be self-companion. Since $\mathcal{B}$ is a self-companion code, then each component of $\mathcal{B}$ contains precisely one plaintext element. Since $g = (g_1, g_2)$ maps the components of $\mathcal{B}$ onto the components of $\mathcal{A}$, then the same holds for the components of $\mathcal{A}$. Hence, $\mathcal{A}$ is a code.

We now define $f = (f_1, f_2) : \mathcal{A} \longrightarrow \mathcal{B}$ on an arbitrary component $X_\mathcal{A} = (P_{X_\mathcal{A}}, C_{X_\mathcal{A}}, e_{X_\mathcal{A}}, d_{X_\mathcal{A}})$ of $\mathcal{A}$. By hypothesis, there is a component $X_\mathcal{B} = (P_{X_\mathcal{B}}, C_{X_\mathcal{B}}, e_{X_\mathcal{B}}, d_{X_\mathcal{B}})$ of $\mathcal{B}$ for which $X_\mathcal{A} = g(X_\mathcal{B})$. We note that $X_\mathcal{A}$ is an isolated vertex if and only if $X_\mathcal{B}$ is as well, and in this case, f must send the vertex in $X_\mathcal{A}$ to the one in $X_\mathcal{B}$. So, we suppose that $X_\mathcal{A}$ and $X_\mathcal{B}$ have both plaintext and codetext elements. We have already seen that they each contain a single plaintext element, say $\pi \in P_{X_\mathcal{A}}$ and $\psi \in P_{X_\mathcal{B}}$. Furthermore, since they are components of self-companion codes, then for each $\kappa \in C_{X_\mathcal{A}}$ and $\chi \in C_{X_\mathcal{B}}$, we must have $(\pi, \kappa) \in e_\mathcal{A} \cap d_\mathcal{A}^{nv}$ and $(\psi, \chi) \in e_\mathcal{B} \cap d_\mathcal{B}^{nv}$. Since $g_1(\psi) = \pi$, we must define $f_1(\pi) = \psi$. For each $\kappa \in C_{X_\mathcal{A}}$, choose a $\kappa_\mathcal{B} \in C_{X_\mathcal{B}}$ for which $g_2(\kappa_\mathcal{B}) = \kappa$ and define $f_2(\kappa) = \kappa_\mathcal{B}$. It is clear that f is a homomorphism and $g \circ f = 1_\mathcal{A}$. Hence, g is a retraction. $\square$

We now give examples to show that the hypotheses of the above theorem cannot be weakened.

Example 6.26. *Let $\mathcal{A} = (\{p_1\}, \{c_1, c_2, c_3\}, \{(p_1, c_1), (p_1, c_2), (p_1, c_3)\}, e_\mathcal{A}^{nv})$ and*

$$\mathcal{B} = (P_\mathcal{B} = \{p_1, p_2\}, C_\mathcal{B} = \{c_1, c_2, c_3\}, e_\mathcal{B} = \{(p_1, c_1), (p_1, c_2), (p_2, c_2), (p_2, c_3)\}, e_\mathcal{B}^{nv})$$

as depicted in Figure 7.
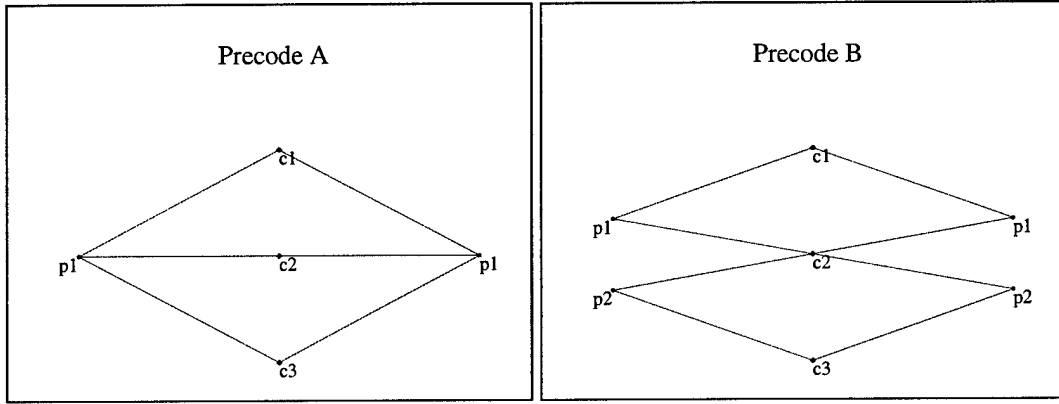
FIGURE 7. The Codomain Is Not a Code

Notice that $\mathcal{B}$ is a self-companion precode which is not a code. Define $g = (g_1, 1_{C_{\mathcal{B}}}) : \mathcal{B} \longrightarrow \mathcal{A}$ via $g_1(p_1) = g_1(p_2) = p_1$. Then g satisfies all hypotheses of the theorem with the exception of $\mathcal{B}$ being a code. However, $\mathcal{B}$ contains no isomorphic copy of $\mathcal{A}$. Thus, by Lemma 6.19, g is not a retraction.

Example 6.27. *Let $\mathcal{A} = (\{p_1, p_2\}, \{c\}, \{(p_1, c), (p_2, c)\}, \{(c, p_1)\})$ and*

$$\mathcal{B} = \mathcal{A}_{\mathcal{M}} = (\{p_1\}, \{_{in}c, p_1 c p_1\}, \{(p_1, p_1 c p_1), (p_1, {}_{in} c), (p_2, {}_{in} c)\}, \{(p_1 c p_1, p_1)\})$$

as depicted in Figure 8. In the picture, we use $p_1 c$ to represent $p_1 c p_1$ and c_in to represent $_{in}c$.
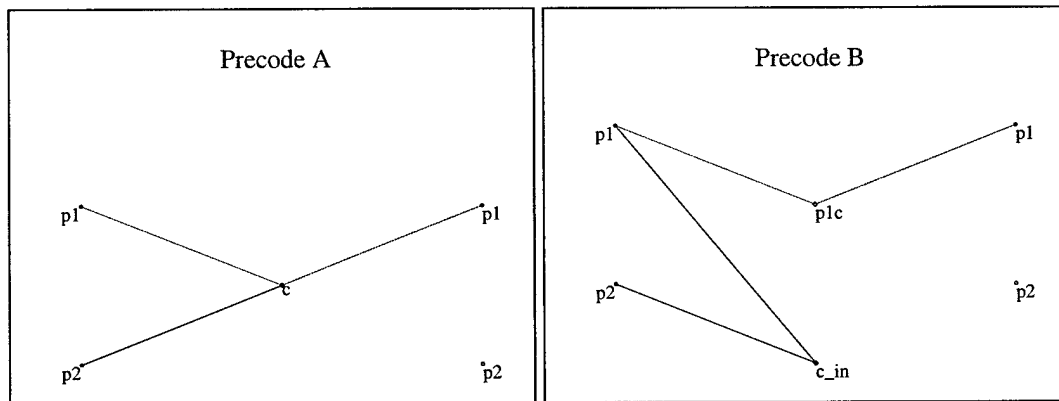
FIGURE 8. The Domain Is a Code Which Is Not Self-Companion

Notice that $\mathcal{B}$ is a code which is not self-companion. The canonical map $\chi : \mathcal{B} \longrightarrow \mathcal{A}$ is not a retraction since $\mathcal{B}$ contains no isomorphic copy of $\mathcal{A}$.

Example 6.28. *We note that in Example 6.27, $\mathcal{A}$ was a precode which was not a code. This example shows that requiring $\mathcal{A}$ to be a code (as in the theorem) does not help either.*

Let

$$\mathcal{A} = (\{p_1\}, \{c_1, c_2, c_3\}, \{(p_1, c_1), (p_1, c_2), (p_1, c_3)\}, \{(c_3, p_1)\})$$

and $\mathcal{B} = (\{p_1, p_2\}, \{c_1, c_2, c_3, c_4, c_5\}, e_{\mathcal{B}}, d_{\mathcal{B}}\}$, where

$$e_{\mathcal{B}} = \{(p_1, c_1), (p_1, c_3), (p_2, c_5), (p_2, c_2)\} \ and \ d_{\mathcal{B}} = \{(c_3, p_1), (c_4, p_1), (c_4, p_2), (c_5, p_2)\},$$

as depicted in Figure 9.
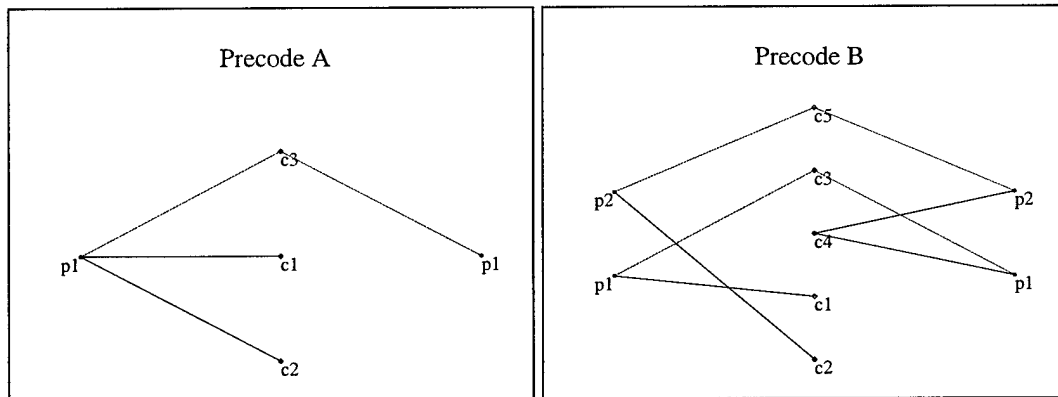
FIGURE 9. The Codomain Is a Code Which Is Not Self-Companion

Notice that $\mathcal{A}$ is a code which is not self-companion. The map $g = (g_1, g_2) : \mathcal{B} \longrightarrow \mathcal{A}$ defined via $g_1(p_1) = g_1(p_2) = p_1$, $g_2(c_1) = c_1$, $g_2(c_2) = c_2$, and $g_2(c_3) = g_2(c_4) = g_2(c_5) = c_3$ is not a retraction since $\mathcal{B}$ contains no isomorphic copy of $\mathcal{A}$.

6.10. **Completeness and Cocompleteness.** Since $\mathfrak{P}$ and $\mathfrak{C}$ have products and equalizers, then Theorem B.49 gives us the following theorem.

Theorem 6.29. *The categories $\mathfrak{P}$ and $\mathfrak{C}$ are complete and have multiple pullbacks, terminal objects, inverse images, finite intersections, intersections of regular subobjects, and inverse limits.*

6.11. **Projective and Injective Precodes.** Recall Definition B.50.

Theorem 6.30. $\mathcal{A}$ *is projective in $\mathfrak{P}$ or $\mathfrak{C}$ if and only if $e_{\mathcal{A}} = \emptyset = d_{\mathcal{A}}$.*

Proof. ($\Rightarrow$) : Suppose that $e_{\mathcal{A}} \cup d_{\mathcal{A}} \neq \emptyset$. W.l.o.g., suppose that $e_{\mathcal{A}} \neq \emptyset$. Let $\mathcal{F} = \mathcal{A}$ and $\mathcal{B} = (P_{\mathcal{A}}, C_{\mathcal{A}}, e_{\mathcal{B}} = \emptyset, d_{\mathcal{B}} = \emptyset)$. Then $g = 1_{\mathcal{A}} : \mathcal{A} \longrightarrow \mathcal{F}$ and $f = 1_{\mathcal{A}} : \mathcal{B} \longrightarrow \mathcal{F}$ are epimorphisms. However, since $e_{\mathcal{A}} \neq \emptyset$ and $e_{\mathcal{B}} = \emptyset$, there is no homomorphism $h : \mathcal{A} \longrightarrow \mathcal{B}$ (much less one for which $f \circ h = g$).

($\Leftarrow$) : Suppose that $e_{\mathcal{A}} = \emptyset = d_{\mathcal{A}}$. Let $\mathcal{B} = (P_{\mathcal{B}}, C_{\mathcal{B}}, e_{\mathcal{B}}, d_{\mathcal{B}})$ and $\mathcal{F} = (P_{\mathcal{F}}, C_{\mathcal{F}}, e_{\mathcal{F}}, d_{\mathcal{F}})$ be precodes (resp. codes). Let $f = (f_1, f_2) : \mathcal{B} \longrightarrow \mathcal{F}$ be an epimorphism and $g = (g_1, g_2) : \mathcal{A} \longrightarrow \mathcal{F}$ be a homomorphism. For every $\pi \in P_{\mathcal{A}}$, we choose $\pi' \in P_{\mathcal{B}}$ such that $\pi' \in f_1^{-1}(g_1(\pi))$, and we define $h_1(\pi) = \pi'$. Similarly, for every $\kappa \in C_{\mathcal{A}}$, we choose $\kappa' \in C_{\mathcal{B}}$ such that $\kappa' \in f_2^{-1}(g_2(\kappa))$, and we define $h_2(\kappa) = \kappa'$. We note that if $P_{\mathcal{A}} = \emptyset$, then $h_1 : P_{\mathcal{A}} \longrightarrow P_{\mathcal{B}}$ and $g_1 : P_{\mathcal{A}} \longrightarrow \mathcal{F}$ are the empty function, and if $C_{\mathcal{A}} = \emptyset$, then $h_2 : C_{\mathcal{A}} \longrightarrow C_{\mathcal{B}}$ and $g_2 : C_{\mathcal{A}} \longrightarrow C_{\mathcal{F}}$ are the empty function. By definition, $f_1 \circ h_1 = g_1$ and $f_2 \circ h_2 = g_2$, so that $f \circ h = g$. Since $e_{\mathcal{A}} = \emptyset = d_{\mathcal{A}}$, then $h = (h_1, h_2) : \mathcal{A} \rightarrow \mathcal{B}$ is a homomorphism. Thus, $\mathcal{A}$ is projective. $\square$

Theorem 6.31. $\mathcal{A}$ *is injective in $\mathfrak{P}$ or $\mathfrak{C}$ if and only if $P_{\mathcal{A}} \neq \emptyset$, $C_{\mathcal{A}} \neq \emptyset$, $e_{\mathcal{A}} = P_{\mathcal{A}} \times C_{\mathcal{A}}$, and $d_{\mathcal{A}} = C_{\mathcal{A}} \times P_{\mathcal{A}}$. Hence, in $\mathfrak{C}$, $\mathcal{A}$ is injective if and only if $\mathcal{A}$ is of the form $\mathcal{A} = (\{\pi\}, C_{\mathcal{A}}, \{\pi\} \times C_{\mathcal{A}}, C_{\mathcal{A}} \times \{\pi\})$.*

Proof. ($\Rightarrow$) : Suppose that it is not the case that $P_{\mathcal{A}} \neq \emptyset$, $C_{\mathcal{A}} \neq \emptyset$, $e_{\mathcal{A}} = P_{\mathcal{A}} \times C_{\mathcal{A}}$, and $d_{\mathcal{A}} = C_{\mathcal{A}} \times P_{\mathcal{A}}$.

Case 1: Suppose that $P_{\mathcal{A}} = \emptyset$ or $C_{\mathcal{A}} = \emptyset$. W.l.o.g., suppose $P_{\mathcal{A}} = \emptyset$. Then $\mathcal{A} = (\emptyset, C_{\mathcal{A}}, \emptyset, \emptyset)$. Note that $\mathcal{B} = (P_{\mathcal{B}} = \{p\}, C_{\mathcal{A}}, \emptyset, \emptyset)$ is a code. Let $\mathcal{F} = \mathcal{A}$, and let $g_1 : P_{\mathcal{F}} \longrightarrow P_{\mathcal{A}}$ and $f_1 : P_{\mathcal{F}} \longrightarrow P_{\mathcal{B}}$

be the empty function. If $C_{\mathcal{A}} = \emptyset$, let $g_2 : C_{\mathcal{F}} \longrightarrow C_{\mathcal{A}}$ and $f_2 : C_{\mathcal{F}} \longrightarrow C_{\mathcal{B}}$ be the empty function; otherwise, let $g_2 = 1_{C_{\mathcal{F}}} = f_2$. In either case, f is a monomorphism. Since $P_{\mathcal{B}} \neq \emptyset$ and $P_{\mathcal{A}} = \emptyset$, there is no homomorphism from $\mathcal{B}$ to $\mathcal{A}$. Thus, $\mathcal{A}$ is not injective.

Case 2: Suppose that $P_{\mathcal{A}} \neq \emptyset \neq C_{\mathcal{A}}$ and that $e_{\mathcal{A}} \neq P_{\mathcal{A}} \times C_{\mathcal{A}}$ or $d_{\mathcal{A}} \neq C_{\mathcal{A}} \times P_{\mathcal{A}}$. W.l.o.g., suppose $e_{\mathcal{A}} \neq P_{\mathcal{A}} \times C_{\mathcal{A}}$. Thus, there exist $\pi \in P_{\mathcal{A}}$ and $\kappa \in C_{\mathcal{A}}$ such that $(\pi, \kappa) \notin e_{\mathcal{A}}$. Notice that $\mathcal{B} = (P_{\mathcal{B}} = \{\pi\}, C_{\mathcal{B}} = \{\kappa\}, e_{\mathcal{B}} = \{(\pi, \kappa)\}, d_{\mathcal{B}} = \{(\kappa, \pi)\})$ and $\mathcal{F} = (\{\pi\}, \{\kappa\}, \emptyset, \emptyset)$ are codes. Also, $g = 1_{\mathcal{F}} : \mathcal{F} \longrightarrow \mathcal{A}$ and $f = 1_{\mathcal{F}} : \mathcal{F} \longrightarrow \mathcal{B}$ are monomorphisms. To have $h \circ f = g$, we must define $h = 1_{\mathcal{B}} : \mathcal{B} \longrightarrow \mathcal{A}$. But, h is not a homomorphism since $(\pi, \kappa) \in e_{\mathcal{B}}$ and $(\pi, \kappa) \notin e_{\mathcal{A}}$.

$(\Leftarrow)$: Suppose that $P_{\mathcal{A}} \neq \emptyset$, $C_{\mathcal{A}} \neq \emptyset$, $e_{\mathcal{A}} = P_{\mathcal{A}} \times C_{\mathcal{A}}$, and $d_{\mathcal{A}} = C_{\mathcal{A}} \times P_{\mathcal{A}}$. Let $\mathcal{B} = (P_{\mathcal{B}}, C_{\mathcal{B}}, e_{\mathcal{B}}, d_{\mathcal{B}})$ and $\mathcal{F} = (P_{\mathcal{F}}, C_{\mathcal{F}}, e_{\mathcal{F}}, d_{\mathcal{F}})$ be precodes (resp. codes). Let $f = (f_1, f_2) : \mathcal{F} \longrightarrow \mathcal{B}$ be a monomorphism and $g = (g_1, g_2) : \mathcal{F} \longrightarrow \mathcal{A}$ be a homomorphism. For all $\pi \in P_{\mathcal{F}}$ and $\kappa \in C_{\mathcal{F}}$, we must set $h_1(f_1(\pi)) = g_1(\pi)$ and $h_2(f_2(\kappa)) = g_2(\kappa)$. This is well-defined since f_1 and f_2 are one-to-one. Now, fix $\pi \in P_{\mathcal{A}}$ and $\kappa \in C_{\mathcal{A}}$. For any $\pi' \in P_{\mathcal{B}}$ such that $\pi' \notin f_1(P_{\mathcal{F}})$, we define $h_1(\pi') = \pi$, and for any $\kappa' \in C_{\mathcal{B}}$ such that $\kappa' \notin f_2(C_{\mathcal{F}})$, we define $h_2(\kappa') = \kappa$. It is clear that $h = (h_1, h_2) : \mathcal{B} \longrightarrow \mathcal{A}$ is a homomorphism since $e_{\mathcal{A}} = P_{\mathcal{A}} \times C_{\mathcal{A}}$ and $d_{\mathcal{A}} = C_{\mathcal{A}} \times P_{\mathcal{A}}$. $\square$

6.12. **Separators and Coseparators.** Recall Definition B.51.

Theorem 6.32. *The categories $\mathfrak{P}$ and $\mathfrak{C}$ have no separators.*

Proof. Consider the codes $\mathcal{A} = (P_{\mathcal{A}} = \{\pi\}, C_{\mathcal{A}} = \emptyset, \emptyset, \emptyset)$ and $\mathcal{B} = (P_{\mathcal{B}} = \{\pi, \psi\}, C_{\mathcal{B}} = \emptyset, \emptyset, \emptyset)$. Define $f_1, g_1 : P_{\mathcal{A}} \longrightarrow P_{\mathcal{B}}$ via $f_1(\pi) = \pi$ and $g_1(\pi) = \psi$. Let $f_2 = g_2 : C_{\mathcal{A}} \longrightarrow C_{\mathcal{B}}$ be the empty function. Then $f = (f_1, f_2), g = (g_1, g_2) : \mathcal{A} \longrightarrow \mathcal{B}$ are distinct precode homomorphisms. If S is a precode with $C_S \neq \emptyset$, then there is no homomorphism $x : S \longrightarrow A$. Thus, for S to be a separator, we must have $C_S = \emptyset$. Similarly, we must have $P_S = \emptyset$. However, the only homomorphism from $S = (\emptyset, \emptyset, \emptyset, \emptyset)$ to $\mathcal{A}$ is the empty homomorphism, $x = (\emptyset, \emptyset)$, and $f \circ x = g \circ x$. Hence, there are no separators. $\square$

Lemma 6.33. *Let $\mathcal{A}$ and $\mathcal{B}$ be codes and let $f = (f_1, h)$ and $g = (g_1, h)$ be distinct homomorphisms from $\mathcal{A}$ to $\mathcal{B}$; that is, $f_1 \neq g_1$. Let S be a code for which there exist $\pi, \psi \in P_S$ with $\pi \neq \psi$ and distinct $\kappa_1, \kappa_2, \kappa_3,$ and $\kappa_4 \in C_S$ such that*

$$\{(\pi, \kappa_1), (\pi, \kappa_2), (\psi, \kappa_2), (\psi, \kappa_4)\} \subseteq e_S \text{ and } \{(\kappa_1, \pi), (\kappa_3, \pi), (\kappa_3, \psi), (\kappa_4, \psi)\} \subseteq d_S.$$

Then there exists a homomorphism $x : \mathcal{B} \longrightarrow S$ satisfying $x \circ f \neq x \circ g$.

Proof. Choose $\pi_{\mathcal{A}} \in P_{\mathcal{A}}$ such that $f_1(\pi_{\mathcal{A}}) \neq g_1(\pi_{\mathcal{A}})$, and set $\pi' = f_1(\pi_{\mathcal{A}}), \psi' = g_1(\pi_{\mathcal{A}})$. For each $\alpha \in P_{\mathcal{B}}$ and each $\beta \in C_{\mathcal{B}}$, define

$$x_1(\alpha) = \begin{cases} \pi, & \text{if } \alpha \neq \psi' \\ \psi, & \text{if } \alpha = \psi' \end{cases} \text{ and } x_2(\beta) = \begin{cases} \kappa_1, & \text{if } \beta \text{ is of type } o \text{ or } (\alpha, \beta) \in e_{\mathcal{B}} \cap d_{\mathcal{B}}^{nv} \text{ for some } \alpha \neq \psi' \\ \kappa_2, & \text{if } \beta \text{ is of type } e \\ \kappa_3, & \text{if } \beta \text{ is of type } d \\ \kappa_4, & \text{if } (\psi', \beta) \in e_{\mathcal{B}} \cap d_{\mathcal{B}}^{nv} \end{cases}$$

We show that $x : \mathcal{B} \longrightarrow S$ is a homomorphism by showing that it preserves the edges in $\mathcal{B}$. Suppose that $(\alpha, \beta) \in e_{\mathcal{B}} \cup d_{\mathcal{B}}^{nv}$. If $(\alpha, \beta) \in e_{\mathcal{B}} \cap d_{\mathcal{B}}^{nv}$, then $x(\alpha, \beta) \in \{(\pi, \kappa_1), (\psi, \kappa_4)\} \in e_S \cap d_S^{nv}$. If $(\alpha, \beta) \in e_{\mathcal{B}} \backslash d_{\mathcal{B}}^{nv}$, then β is of type e and $x(\alpha, \beta) \in \{(\pi, \kappa_2), (\psi, \kappa_2)\} \in e_S$. If $(\alpha, \beta) \in d_{\mathcal{B}}^{nv} \backslash e_{\mathcal{B}}$, then β is of type d and $x(\alpha, \beta) \in \{(\pi, \kappa_3), (\psi, \kappa_3)\} \in d_S^{nv}$. Since

$$x_1(f_1(\pi_{\mathcal{A}})) = x_1(\pi') = \pi \neq \psi = x_1(\psi') = x_1(g_1(\pi_{\mathcal{A}})),$$

then $x \circ f \neq x \circ g$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

Theorem 6.34. *Let $S = (P_S, C_S, e_S, d_S)$ be a precode, and consider the following conditions on S:*

i) There exist distinct $\pi, \psi \in P_S$ with $\pi \neq \psi$ and $\kappa \in C_S$ such that $\{(\pi, \kappa), (\psi, \kappa)\} \subseteq e_S \cap d_S^{nv}$.

ii) There exist $\pi \in P_S$ and $\kappa, \chi \in C_S$ with $\kappa \neq \chi$ such that $\{(\pi, \kappa), (\pi, \chi)\} \subseteq e_S \cap d_S^{nv}$.

iii) There exist $\pi, \psi \in P_S$ with $\pi \neq \psi$ and distinct $\kappa_1, \kappa_2, \kappa_3$, and $\kappa_4 \in C_S$ such that

$$\{(\pi, \kappa_1), (\pi, \kappa_2), (\psi, \kappa_2), (\psi, \kappa_4)\} \subseteq e_S \text{ and } \{(\kappa_1, \pi), (\kappa_3, \pi), (\kappa_3, \psi), (\kappa_4, \psi)\} \subseteq d_S.$$

S is a coseparator in the category of precodes, $\mathfrak{P}$, if and only if conditions (i) and (ii) hold. If S is a code, then it is a coseparator in the category of codes, $\mathfrak{C}$, if and only if conditions (ii) and (iii) hold.

Proof. $(\Rightarrow)$: Suppose S is a coseparator in $\mathfrak{P}$. We show that (i) holds. Let

$$\mathcal{A} = (P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A}, d_\mathcal{A}) = (\{\pi'\}, \{\kappa'\}, \{(\pi', \kappa')\}, \{(\kappa', \pi')\})$$

and

$$\mathcal{B} = (P_\mathcal{B}, C_\mathcal{B}, e_\mathcal{B}, d_\mathcal{B}) = (\{\pi', \psi'\}, \{\kappa'\}, \{(\pi', \kappa'), (\psi', \kappa')\}, \{(\kappa', \pi'), (\kappa', \psi')\}).$$

Define $f_1, g_1 : P_\mathcal{A} \longrightarrow P_\mathcal{B}$ via $f_1(\pi') = \pi'$ and $g_1(\pi') = \psi'$. Then $f = (f_1, 1_{C_\mathcal{A}}), g = (g_1, 1_{C_\mathcal{A}}) :$ $\mathcal{A} \longrightarrow \mathcal{B}$ are distinct precode homomorphisms. Since S is a coseparator, there exists a homomorphism $x = (x_1, x_2) : B \longrightarrow S$ with $x_1 \circ f_1 \neq x_1 \circ g_1$, so that $x_1(\pi') \neq x_1(\psi')$. Since $(\pi', \kappa'), (\psi', \kappa') \in e_\mathcal{B} \cap d_\mathcal{B}^{nv}$, then $(x_1(\pi'), x_2(\kappa')), (x_1(\psi'), x_2(\kappa')) \in e_S \cap d_S^{nv}$. Thus, (i) holds.

Suppose now that S is a coseparator in $\mathfrak{C}$. We show that (iii) holds. Let

$$\mathcal{A} = (P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A}, d_\mathcal{A}) = (\{\pi'\}, \{\kappa'\}, \{(\pi', \kappa')\}, \emptyset) \text{ and } \mathcal{B} = (P_\mathcal{B}, C_\mathcal{B}, e_\mathcal{B}, d_\mathcal{B}),$$

where $P_\mathcal{B} = (\{\pi', \psi'\}$, $C_\mathcal{B} = \{\kappa_1', \kappa_2', \kappa_3', \kappa_4'\}$, $e_\mathcal{B} = \{(\pi', \kappa_1'), (\pi', \kappa_2'), (\psi', \kappa_2'), (\psi', \kappa_4')\}$, and $d_\mathcal{B} = \{(\kappa_1', \pi'), (\kappa_3', \pi'), (\kappa_3', \psi'), (\kappa_4', \psi')\})$. Define $f_1, g_1 : P_\mathcal{A} \longrightarrow P_\mathcal{B}$ via $f_1(\pi') = \pi'$ and $g_1(\pi') = \psi'$. Let $h : C_\mathcal{A} \longrightarrow C_\mathcal{B}$ be the constant function onto κ_2'. Then $f = (f_1, h), g = (g_1, h) : \mathcal{A} \longrightarrow \mathcal{B}$ are distinct precode homomorphisms. Since S is a coseparator, there exists a homomorphism $x = (x_1, x_2) : B \longrightarrow S$ such that $x_1 \circ f_1 \neq x_1 \circ g_1$. Thus, $x_1(\pi') \neq x_1(\psi')$.

Set $\pi = x_1(\pi')$, $\psi = x_1(\psi')$, $\kappa_1 = x_2(\kappa_1')$, $\kappa_2 = x_2(\kappa_2')$, $\kappa_3 = x_2(\kappa_3')$, and $\kappa_4 = x_2(\kappa_4')$.

Since $(\pi', \kappa_1'), (\psi', \kappa_4') \in e_\mathcal{B} \cap d_\mathcal{B}^{nv}$ and since x is a homomorphism, then $(\pi, \kappa_1), (\psi, \kappa_4) \in e_S \cap d_S^{nv}$. Since S is a code, and $\pi \neq \psi$, then κ_1 and κ_4 must be distinct. Similarly, $(\pi', \kappa_2'), (\psi', \kappa_2') \in e_\mathcal{B}$ and

$(\kappa_3', \pi'), (\kappa_3', \psi') \in d_{\mathcal{B}}$, imply that $(\pi, \kappa_2), (\psi, \kappa_2) \in e_S$ and $(\kappa_3, \pi), (\kappa_3, \psi) \in d_S$. Since S is a code, and $\pi \neq \psi$, then κ_2 and κ_3 must be distinct. It is also clear that κ_2 and κ_3 must be distinct from κ_1 and κ_4. Thus, (iii) holds.

We now show that if S is a coseparator in either category, then (ii) holds. Consider the codes

$$\mathcal{A} = (P_{\mathcal{A}}, C_{\mathcal{A}}, e_{\mathcal{A}}, d_{\mathcal{A}}) = (\{\pi'\}, \{\kappa'\}, \{(\pi', \kappa')\}, \{(\kappa', \pi')\})$$

and

$$\mathcal{B} = (P_{\mathcal{B}}, C_{\mathcal{B}}, e_{\mathcal{B}}, d_{\mathcal{B}}) = (\{\pi'\}, \{\kappa', \chi'\}, \{(\pi', \kappa'), (\pi', \chi')\}, \{(\kappa', \pi'), (\chi', \pi')\}).$$

Define $f_2, g_2 : C_{\mathcal{A}} \longrightarrow C_{\mathcal{B}}$ via $f_2(\kappa') = \kappa'$ and $g_2(\kappa') = \chi'$. Then $f = (1_{P_{\mathcal{A}}}, f_2), g = (1_{P_{\mathcal{A}}}, g_2) : \mathcal{A} \longrightarrow \mathcal{B}$ are distinct precode homomorphisms. Since S is a coseparator, there is a homomorphism $x = (x_1, x_2) : B \longrightarrow S$ with $x_2 \circ f_2 \neq x_2 \circ g_2$. Thus,(ii) holds since $x_2(\kappa') \neq x_2(\chi')$ and $(x_1(\pi'), x_2(\kappa')), (x_1(\pi'), x_2(\chi')) \in e_S \cap d_S^{nv}$.

($\Leftarrow$) : Suppose S satisfies (i) and (ii). Let $\mathcal{A}$ and $\mathcal{B}$ be precodes and $f = (f_1, f_2), g = (g_1, g_2) : \mathcal{A} \longrightarrow \mathcal{B}$ be distinct precode homomorphisms.

Case 1: Suppose that $f_2 \neq g_2$. Thus, there exists $\beta \in C_{\mathcal{A}}$ for which $\kappa' = f_2(\beta) \neq g_2(\beta) = \chi'$. Let $\pi \in P_S$ and $\kappa, \chi \in C_S$ with $\kappa \neq \chi$ be as in condition (ii). Define $x_1 : P_{\mathcal{B}} \longrightarrow P_S$ via $x_1(\pi') = \pi$ for all $\pi' \in P_{\mathcal{B}}$ and $x_2 : C_{\mathcal{B}} \longrightarrow C_S$ via

$$x_2(\beta') = \begin{cases} \kappa, & \text{if } \beta' = \kappa' \\ \chi, & \text{if } \beta' \neq \kappa' \end{cases}$$

Since $(\pi, \kappa), (\pi, \chi) \in e_S \cap d_S^{nv}$, then $x = (x_1, x_2) : B \longrightarrow S$ is a homomorphism. Also, $x \circ f \neq x \circ g$ since $x_2(f_2(\beta)) = x_2(\kappa') = \kappa \neq \chi = x_2(\chi') = x_2(g_2(\beta))$. Hence, S is a coseparator in $\mathfrak{P}$.

Case 2: Suppose that $f_1 \neq g_1$. Thus, there exists $\alpha \in P_{\mathcal{A}}$ for which $\pi' = f_1(\alpha) \neq g_1(\alpha) = \psi'$. Let $\pi, \psi \in P_S$ with $\pi \neq \psi$ and $\kappa \in C_S$ be as in condition (i). Define $x_2 : C_{\mathcal{B}} \longrightarrow C_S$ via $x_2(\kappa') = \kappa$ for all $\kappa' \in C_{\mathcal{B}}$ and $x_1 : P_{\mathcal{B}} \longrightarrow P_S$ via

$$x_1(\alpha') = \begin{cases} \pi, & \text{if } \alpha' = \pi' \\ \\ \psi, & \text{if } \alpha' \neq \pi' \end{cases}$$

Since $(\pi, \kappa), (\psi, \kappa) \in e_S \cap d_S^{nv}$, then $x = (x_1, x_2) : B \longrightarrow S$ is a precode homomorphism. Also, $x \circ f \neq x \circ g$ since $x_1(f_1(\alpha)) = x_1(\pi') = \pi \neq \psi = x_1(\psi') = x_1(g_1(\alpha))$. Thus, S is a coseparator in $\mathfrak{P}$.

If S is a code for which conditions (ii) and (iii) hold, we suppose that $f = (f_1, f_2), g = (g_1, g_2) : \mathcal{A} \longrightarrow \mathcal{B}$ are distinct homomorphisms between codes. If $f_2 \neq g_2$, then the proof above for precodes works here too by employing (ii). If $f_2 = g_2$ and $f_1 \neq g_1$, then Lemma 6.33 guarantees a homomorphism $x : B \longrightarrow S$ satisfying $x \circ f \neq x \circ g$. Hence, S is a coseparator in $\mathfrak{C}$. $\qquad \square$

In Algorithm 7.2, we show how to construct the split of a precode. It is interesting to note that the subcode described in condition (iii) in Theorem 6.34 is the split of the subprecode given by condition (i). Furthermore, since the subprecode described in condition (ii) is a code, it is its split as well. Thus, we may restate the theorem as follows:

Theorem 6.35. *S is a coseparator in $\mathfrak{P}$ if and only if it contains*

$$\mathcal{A} = (\{\pi, \psi\}, \{\kappa\}, \{(\pi, \kappa), (\psi, \kappa)\}, \{(\kappa, \pi), (\kappa, \psi)\})$$

and

$$\mathcal{B} = (\{\pi'\}, \{\kappa', \chi'\}, \{(\pi', \kappa'), (\pi', \chi')\}, \{(\kappa', \pi'), (\chi', \pi')\})$$

as subprecodes. If S is a code, then it is a coseparator in $\mathfrak{C}$ if and only if it contains $\mathcal{A}_{\mathsf{M}}$ and $\mathcal{B}_{\mathsf{M}}$ as subcodes.

7. SPLITTING A PRECODE

7.1. The ED-Split of a Precode.
We start with a method of splitting $\mathcal{A}$ which will be useful when we discuss precode parametrizations in Section 8.

Algorithm 7.1. *Let $\mathcal{A}$ be a precode. We construct the precode*

$$\mathcal{A}_{|edl} = (P_{\mathcal{A}_{|edl}} = P_{\mathcal{A}}, C_{\mathcal{A}_{|edl}}, e_{\mathcal{A}_{|edl}}, d_{\mathcal{A}_{|edl}}),$$

*called the **ed-split** of $\mathcal{A}$, as follows:*

Set $P_{\mathcal{A}_{|edl}} = P_{\mathcal{A}}$, $C_{\mathcal{A}_{|edl}} = \emptyset$, $e_{\mathcal{A}_{|edl}} = \emptyset$, and $d_{\mathcal{A}_{|edl}} = \emptyset$.

For each $\kappa \in C_{\mathcal{A}}$

  If $\exists \pi_1, \pi_2 \in P_{\mathcal{A}}$ such that $\pi_1 \neq \pi_2$, $(\pi_1, \kappa) \in e_{\mathcal{A}}$, and $(\pi_2, \kappa) \in d_{\mathcal{A}}^{nv}$, then

    /* κ must be split */

    Add $_{in}\kappa$ to $C_{\mathcal{A}_{|edl}}$

    Add κ_{out} to $C_{\mathcal{A}_{|edl}}$

    For each $\pi \in P_{\mathcal{A}}$ such that $(\pi, \kappa) \in e_{\mathcal{A}}$

      Add $(\pi, _{in}\kappa)$ to $e_{\mathcal{A}_{|edl}}$

    end for

    For each $\pi \in P_{\mathcal{A}}$ such that $(\pi, \kappa) \in d_{\mathcal{A}}^{nv}$

      Add (π, κ_{out}) to $d_{\mathcal{A}_{|edl}}^{nv}$

    end for

  else

    /* κ does not need to be split */

    Add κ to $C_{\mathcal{A}_{|edl}}$

    For each $\pi \in P_{\mathcal{A}}$

      If $(\pi, \kappa) \in e_{\mathcal{A}}$, then

        Add (π, κ) to $e_{\mathcal{A}_{|edl}}$

        end if

      If $(\pi, \kappa) \in d_A^{nv}$, then

        Add (π, κ) to $d_{A_{|\text{edl}}}^{nv}$

      end if

    end for

  end if

end for

7.2. The Split of a Precode.
The following "split" is in many ways the most useful.

Algorithm 7.2. *Let A be a precode. We construct a code A_{M}, called the **split of** A, as follows:*

Set $P_{A_{\text{M}}} = P_A$, $C_{A_{\text{M}}} = \emptyset$, $e_{A_{\text{M}}} = \emptyset$, and $d_{A_{\text{M}}} = \emptyset$.

For each $\kappa \in C_A$

  If $\exists \pi_1, \pi_2 \in P_A$ such that $\pi_1 \neq \pi_2$, $(\pi_1, \kappa) \in e_A$, and $(\pi_2, \kappa) \in d_A^{nv}$, then

    /* κ must be split */

    For each $\pi \in P_A$ such that $(\pi, \kappa) \in e_A \cap d_A^{nv}$

      Add $\pi \kappa \pi$ to $C_{A_{\text{M}}}$

      Add $(\pi, \pi \kappa \pi)$ to $e_{A_{\text{M}}}$

      Add $(\pi, \pi \kappa \pi)$ to $d_{A_{\text{M}}}^{nv}$

    end for

  If $\exists \pi_1 \in P_A$ such that $(\pi_1, \kappa) \in e_A \backslash d_A^{nv}$ or

    if $\exists \pi_1, \pi_2 \in P_A$ such that $\pi_1 \neq \pi_2$ and $(\pi_1, \kappa), (\pi_2, \kappa) \in e_A$, then

    Add ${}_{in}\kappa$ to $C_{A_{\text{M}}}$

    For each $\pi \in P_A$ such that $(\pi, \kappa) \in e_A$

      Add $(\pi, {}_{in}\kappa)$ to $e_{A_{\text{M}}}$

    end for

end if

If $\exists \pi_1 \in P_A$ such that $(\pi_1, \kappa) \in d_A^{nv} \backslash e_A$ or

    if $\exists \pi_1, \pi_2 \in P_A$ such that $\pi_1 \neq \pi_2$ and $(\pi_1, \kappa), (\pi_2, \kappa) \in d_A^{nv}$, then

    Add κ_{out} to C_{A_M}

    For each $\pi \in P_A$ such that $(\pi, \kappa) \in d_A^{nv}$

        Add (π, κ_{out}) to $d_{A_M}^{nv}$

    end for

    end if

else

    /* κ does not need to be split */

    Add κ to C_{A_M}

    For each $\pi \in P_A$

        If $(\pi, \kappa) \in e_A$, then

            Add (π, κ) to e_{A_M}

        end if

        If $(\pi, \kappa) \in d_A^{nv}$, then

            Add (π, κ) to $d_{A_M}^{nv}$

        end if

    end for

    end if

end for

We can give an alternate description of Algorithm 7.2 using the synoptic codebook matrices M_A and M_{A_M}. Let $M_A(\kappa)(\pi)$ denote $M_A(\pi, \kappa)$, the π-th entry of the κ-th column of M_A. In the following algorithm, we use a question mark as a wildcard character. That is, we use 1? to represent either symbol in $\{e = 10, s = 11\}$ and ?1 to represent either symbol in $\{d = 01, s = 11\}$.

Algorithm 7.3. *Let $\mathcal{A}$ be a precode. We construct $M_{\mathcal{A}_{\text{VI}}}$ from $M_{\mathcal{A}}$ as follows:*

Set $P_{\mathcal{A}_{\text{VI}}} = P_{\mathcal{A}}$, $C_{\mathcal{A}_{\text{VI}}} = \emptyset$, $e_{\mathcal{A}_{\text{VI}}} = \emptyset$, and $d_{\mathcal{A}_{\text{VI}}} = \emptyset$.

For each κ in $C_{\mathcal{A}}$

   If $M_{\mathcal{A}}(\kappa)(\pi) = 1?$ and $M_{\mathcal{A}}(\kappa)(\psi) =?1$ for some $\pi, \psi \in P_{\mathcal{A}}$ such that $\pi \neq \psi$, then

      /* κ must be split */

      For each $\psi \in P_{\mathcal{A}}$ such that $M_{\mathcal{A}}(\kappa)(\psi) = 11$

         Add a column to $M_{\mathcal{A}_{\text{VI}}}$ with a 11 in row ψ and 00's elsewhere.

      end for

      If $M_{\mathcal{A}}(\kappa)(\pi) = 10$ for some $\pi \in P_{\mathcal{A}}$ or

         if $M_{\mathcal{A}}(\kappa)(\pi) = 1?$ and $M_{\mathcal{A}}(\kappa)(\psi) = 1?$ for some $\pi, \psi \in P_{\mathcal{A}}$ such that $\pi \neq \psi$, then

         Add a column to $M_{\mathcal{A}_{\text{VI}}}$ with a 10 in each row for which there is a 1? in $M_{\mathcal{A}}(\kappa)$

            and with 00's elsewhere

      end if

      If $M_{\mathcal{A}}(\kappa)(\pi) = 01$ for some $\pi \in P_{\mathcal{A}}$ or

         if $M_{\mathcal{A}}(\kappa)(\pi) =?1$ and $M_{\mathcal{A}}(\kappa)(\psi) =?1$ for some $\pi, \psi \in P_{\mathcal{A}}$ such that $\pi \neq \psi$, then

         Add a column to $M_{\mathcal{A}_{\text{VI}}}$ with a 01 in each row for which there is a ?1 in $M_{\mathcal{A}}(\kappa)$

            and with 00's elsewhere

      end if

   else

      /* κ does not need to be split */

      Add a column to $M_{\mathcal{A}_{\text{VI}}}$ identical to $M_{\mathcal{A}}(\kappa)$

   end if

end for

Remark 7.4. *In constructing $\mathcal{A}_{\text{VI}}$, for each $\kappa \in C_{\mathcal{A}}$, we add to $C_{\mathcal{A}_{\text{VI}}}$ some nonempty subset $K_\kappa \subseteq (\{_{in}\kappa, \kappa_{out}, \kappa\} \cup \bigcup_{\pi \in P_{\mathcal{A}}} \{\pi\kappa\pi\})$. Furthermore, $P_{\mathcal{A}_{\text{VI}}} = P_{\mathcal{A}}$, and $(\pi, \kappa) \in e_{\mathcal{A}}$ if and only if*

$(\pi, k) \in e_{A_{\mathcal{M}}}$ *for some* $k \in K_\kappa$. *Similarly,* $(\kappa, \pi) \in d_A$ *if and only if* $(k, \pi) \in d_{A_{\mathcal{M}}}$ *for some* $k \in K_\kappa$.

Hence, there is a canonical strong epimorphism $k = (k_1, k_2) : A_{\mathcal{M}} \to A$ *such that* $k_1 = 1_{P_A}$ *and for*

each $\kappa \in C_A$, $k_2(K_\kappa) = \{\kappa\}$. *We say that* $\kappa \in C_A$ *is a **split vertex** if* $|K_\kappa = k_2^{-1}(\kappa)| > 1$.

We next state and prove a theorem which is an analogue of Theorem 7 in [4] for $A_{\mathcal{M}}$. We first

define several sets used in the proof of the theorem and the discussion which follows it.

Notation 7.5. *Let* $h = (h_1, h_2) : \hat{A} \to A$ *be a precode homomorphism between the precodes* $\hat{A}$ *and*

A. *Let* $\hat{\kappa} \in C_{\hat{A}}$ *and* $\kappa = h_2(\hat{\kappa})$. *We define*

$$\hat{P}_{\hat{e}}(\hat{\kappa}) = \{\hat{\pi} \in P_{\hat{A}} \,|\, (\hat{\pi}, \hat{\kappa}) \in e_{\hat{A}}\}$$

$$\hat{P}_{\hat{d}^{nv}}(\hat{\kappa}) = \{\hat{\pi} \in P_{\hat{A}} \,|\, (\hat{\pi}, \hat{\kappa}) \in d_{\hat{A}}^{nv}\}$$

$$P_e(\kappa) = \{\pi \in P_A \,|\, (\pi, \kappa) \in e_A\}$$

$$P_{d^{nv}}(\kappa) = \{\pi \in P_A \,|\, (\pi, \kappa) \in d_A^{nv}\}$$

$$\hat{E}(\hat{\kappa}) = (P_{\hat{A}} \times \{\hat{\kappa}\}) \cap e_{\hat{A}} = \hat{P}_{\hat{e}}(\hat{\kappa}) \times \{\hat{\kappa}\} \subseteq e_{\hat{A}}$$

$$\hat{D}^{nv}(\hat{\kappa}) = (P_{\hat{A}} \times \{\hat{\kappa}\}) \cap d_{\hat{A}}^{nv} = \hat{P}_{\hat{d}^{nv}}(\hat{\kappa}) \times \{\hat{\kappa}\} \subseteq d_{\hat{A}}^{nv}$$

$$E(\kappa) = P_e(\kappa) \times \{\kappa\} \subseteq e_A.$$

$$D^{nv}(\kappa) = P_{d^{nv}}(\kappa) \times \{\kappa\} \subseteq d_A^{nv}.$$

$$E_h(\hat{\kappa}) = h_1(\hat{P}_{\hat{e}}(\hat{\kappa})) \times \{\kappa\} \subseteq E(\kappa) \subseteq e_A$$

$$D_h^{nv}(\hat{\kappa}) = h_1(\hat{P}_{\hat{d}^{nv}}(\hat{\kappa})) \times \{\kappa\} \subseteq D^{nv}(\kappa) \subseteq d_A^{nv}$$

Remark 7.6. *In plain language, we have the following descriptions of the sets defined above:*

$\hat{P}_{\hat{e}}(\hat{\kappa})$ *is the set of elements in* $P_{\hat{A}}$ *which are adjacent to* $\hat{\kappa}$ *via an edge in* $e_{\hat{A}}$.

$\hat{P}_{\hat{d}^{nv}}(\hat{\kappa})$ *is the set of elements in* $P_{\hat{A}}$ *which are adjacent to* $\hat{\kappa}$ *via an edge in* $d_{\hat{A}}^{nv}$.

$P_e(\kappa)$ *is the set of elements in* P_A *which are adjacent to* κ *via an edge in* e_A.

$P_{d^{nv}}(\kappa)$ *is the set of elements in* P_A *which are adjacent to* κ *via an edge in* d_A^{nv}.

$\hat{E}(\hat{\kappa})$ *is the set of edges in* $e_{\hat{A}}$ *which are incident on* $\hat{\kappa}$.

$\hat{D}^{nv}(\hat{\kappa})$ *is the set of edges in* $d_{\hat{A}}^{nv}$ *which are incident on* $\hat{\kappa}$.

$E(\kappa)$ *is the set of edges in* e_A *which are incident on* κ.

$D^{nv}(\kappa)$ is the set of edges in d_A^{nv} which are incident on κ.

$E_h(\hat{\kappa})$ is the set of edges in e_A which are the images under h of the edges in $\hat{E}(\hat{\kappa})$.

$D_h^{nv}(\hat{\kappa})$ is the set of edges in d_A^{nv} which are the images under h of the edges in $\hat{D}^{nv}(\hat{\kappa})$.

Theorem 7.7. *Let A be a precode with associated split code $A_{|1}$ and canonical strong epimorphism $k = (k_1, k_2) : A_{|1} \to A$. Let $\hat{A}$ be a **code** and $h = (h_1, h_2) : \hat{A} \to A$ be a precode homomorphism. Then there exists a precode homomorphism $f = (f_1, f_2) : \hat{A} \to A_{|1}$ such that $h = k \circ f$.*

Proof. We recall that by definition, $P_{A_{|1}} = P_A$ and $k_1 = 1_{P_A}$. Since we need $k_1 \circ f_1 = h_1$, then we must set $f_1 = h_1$. Let $\hat{\kappa} \in C_{\hat{A}}$ and $\kappa = h_2(\hat{\kappa}) \in C_A$. We show how to define $f_2(\hat{\kappa})$ using the sets defined in Notation 7.5.

Case 1: Suppose that $\hat{\kappa} \in J(\hat{A}) = RAN(e_{\hat{A}}) \cap DOM(d_{\hat{A}})$. Since $\hat{A}$ is a code, there is some $\hat{\pi} \in P_{\hat{A}}$ such that $(\hat{\pi}, \hat{\kappa}) \in e_{\hat{A}} \cap d_{\hat{A}}^{nv}$. Furthermore, $(\hat{\pi}, \hat{\kappa})$ is the only edge in $e_{\hat{A}}$ incident to $\hat{\kappa}$ and $(\hat{\kappa}, \hat{\pi})$ is the only edge in $d_{\hat{A}}$ incident from $\hat{\kappa}$. Since h is a homomorphism, then $(\pi = h_1(\hat{\pi}), \kappa)$ must be in $e_A \cap d_A^{nv}$. As in Algorithm 7.2, there is exactly one codetext vertex $b \in K_\kappa = k_2^{-1}(\kappa)$ such that $(\pi, b) \in e_{A_{|1}} \cap d_{A_{|1}}^{nv}$. In particular, $b = \kappa$ if κ is not a split vertex, and $b = \pi \kappa \pi$ if κ is a split vertex. In either case, we must set $f_2(\hat{\kappa}) = b$.

Case 2: Suppose that $\hat{\kappa} \in C_{\hat{A}} \backslash J(\hat{A})$. Thus, either $\hat{E}(\hat{\kappa}) = \emptyset$ or $\hat{D}^{nv}(\hat{\kappa}) = \emptyset$. That is, any edges in $\hat{A}$ incident on $\hat{\kappa}$ must be contained in one of $e_{\hat{A}}$ or $d_{\hat{A}}^{nv}$. W.l.o.g., suppose that $\hat{D}^{nv}(\hat{\kappa}) = \emptyset$.

Case 2.1: Suppose κ is not a split vertex. Then $k_2^{-1}(\kappa) = \{\kappa\}$, and we must define $f_2(\hat{\kappa}) = \kappa$.

Case 2.2: Suppose κ is a split vertex. Then $\kappa \in J(A)$ and $|k_2^{-1}(\kappa)| > 1$.

Case 2.2.1: Suppose $|E_h(\hat{\kappa})| > 1$. Then $|\hat{E}(\hat{\kappa})| > 1$. Since κ is a split vertex, then $_{in}\kappa \in k_2^{-1}(\kappa)$, and it is the only vertex in $k_2^{-1}(\kappa)$ which has more than one edge in $e_{A_{|1}}$ incident to it. Thus, we must define $f_2(\hat{\kappa}) =_{in} \kappa$.

Case 2.2.2: Suppose $|E_h(\hat{\kappa})| = 1$; that is, $E_h(\hat{\kappa}) = \{(\pi, \kappa)\}$ for some $\pi \in P_A$.

Case 2.2.2.1: Suppose $(\pi, \kappa) \in e_{\mathcal{A}} \setminus (e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv})$. Then $\pi \kappa \pi \notin k_2^{-1}(\kappa)$. Thus, $_{in}\kappa \in k_2^{-1}(\kappa)$, and it is the only vertex in $k_2^{-1}(\kappa)$ which has an edge in $e_{\mathcal{A}_{\mathcal{M}}}$ incident from π to it. Hence, we must define $f_2(\hat{\kappa}) =_{in} \kappa$.

Case 2.2.2.2: Suppose $(\pi, \kappa) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$.

Case 2.2.2.2.1: Suppose $|E(\kappa)| \leq 1$. Since $|E_h(\hat{\kappa})| = 1$ and $E_h(\hat{\kappa}) \subseteq E(\kappa)$, then $E(\kappa) = E_h(\hat{\kappa}) = \{(\pi, \kappa)\}$. Since $(\pi, \kappa) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$, then $k_2^{-1}(\kappa) = \{\pi\kappa\pi, \kappa_{out}\}$, and $\pi\kappa\pi$ is the only vertex in $k_2^{-1}(\kappa)$ which has an edge in $e_{\mathcal{A}_{\mathcal{M}}}$ incident from π to it. So, we must set $f_2(\hat{\kappa}) = \pi\kappa\pi$.

Case 2.2.2.2.2: Suppose $|E(\kappa)| > 1$. Since $(\pi, \kappa) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$, then $k_2^{-1}(\kappa) \supseteq \{_{in}\kappa, \pi\kappa\pi\}$, and these are the only vertices in $k_2^{-1}(\kappa)$ which have edges in $e_{\mathcal{A}_{\mathcal{M}}}$ incident from π to them. Since $|E_h(\hat{\kappa})| = 1$, we may either set $f_2(\hat{\kappa}) = \pi\kappa\pi$ or $f_2(\hat{\kappa}) =_{in} \kappa$.

Case 2.2.3: Suppose $|E_h(\hat{\kappa})| = 0$. Then $|\hat{E}(\hat{\kappa})| = 0$ (i.e. $\hat{\kappa}$ is an isolated vertex), and we may set $f_2(\hat{\kappa}) = m$, for any $m \in k_2^{-1}(\kappa)$. Since κ is a split vertex, then $|k_2^{-1}(\kappa)| \geq 2$. $\square$

Remark 7.8. *We note that the factorization in Theorem 7.7 is unique as long as Cases 2.2.2.2.2 and 2.2.3 never apply. Thus, the precode homomorphism $f = (f_1, f_2) : \hat{A} \to \mathcal{A}_{\mathcal{M}}$ is unique unless $C_{\hat{A}}$ contains a vertex $\hat{\kappa}$ such that $|k_2^{-1}(h_2(\hat{\kappa}))| > 1$ and any of the conditions specified below in (a),(b), or (c) hold:*

(a) $\hat{E}(\hat{\kappa}) = \emptyset = \hat{D}^{nv}(\hat{\kappa})$ (Case 2.2.3)

(b) $\hat{D}^{nv}(\hat{\kappa}) = \emptyset$, $|E_h(\hat{\kappa})| = 1$, $E_h(\hat{\kappa}) \subseteq e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$, and $|E(\kappa)| > 1$ (Case 2.2.2.2.2)

(c) $\hat{E}(\hat{\kappa}) = \emptyset$, $|D_h^{nv}(\hat{\kappa})| = 1$, $D_h^{nv}(\hat{\kappa}) \subseteq e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$, and $|D^{nv}(\kappa)| > 1$ (Case 2.2.2.2.2)

In graph terms, f is unique unless there exists $\hat{\kappa} \in C_{\hat{A}}$ such that $h_2(\hat{\kappa})$ is a split vertex and either

(a) $\hat{\kappa}$ is isolated; i.e., $e_{\hat{A}}^{nv}(\hat{\kappa}) = \emptyset$ and $d_{\hat{A}}(\hat{\kappa}) = \emptyset$ (Case 2.2.3) OR

(b) the following three conditions hold (Case 2.2.2.2.2):

(i) all the edges incident on $\hat{\kappa}$ are contained in $e_{\hat{A}}$ (respectively $d_{\hat{A}}^{nv}$)

(ii) the image under h of these edges is a single edge which is contained in $e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$

(iii) there is more than one edge in $e_{\mathcal{A}}$ (respectively $d_{\mathcal{A}}^{nv}$) incident on κ.

Notice that there is no set of conditions on $\hat{A}$ alone which will guarantee uniqueness.

From a categorical perspective, we would like to have a result analogous to Theorem 7 in [4] to hold for the split operator. The above theorem shows that no such result holds for the entire category of precodes. We now attempt to construct a subcategory for which we have both existence and uniqueness of factorization. However, per the above remark, we cannot merely place restrictions on the structure of $\hat{A}$. We must restrict the homomorphisms contained in a candidate subcategory. We begin with the largest potential subcategory. That is, we exclude from consideration all homomorphisms prohibited by the above remark. Consider the following example.

Example 7.9. *We define*

$$\hat{A} = (P_{\hat{A}} = \{1,2\}, C_{\hat{A}} = \{\alpha, \beta, \kappa\}, e_{\hat{A}} = \{(1,\beta),(2,\kappa)\}, d_{\hat{A}} = \{(\alpha,1)\}),$$

$$\bar{A} = (P_{\bar{A}} = \{1,2\}, C_{\bar{A}} = \{\alpha, \kappa\}, e_{\bar{A}} = \{(1,\alpha),(2,\kappa)\}, d_{\bar{A}} = \{(\alpha,1)\}),$$

and

$$A = (P_A = \{1,2\}, C_A = \{\alpha\}, e_A = \{(1,\alpha),(2,\alpha)\}, d_A = \{(\alpha,1)\}).$$

Define $f = (1_{P_A}, f_2) : \hat{A} \longrightarrow \bar{A}$ and $g = (1_{P_A}, g_2) : \bar{A} \longrightarrow A$ by defining

$$f_2(\kappa) = \kappa, f_2(\alpha) = f_2(\beta) = \alpha, \text{ and } g_2(\alpha) = g_2(\kappa) = \alpha.$$

It is clear that $\hat{A}$ and $\bar{A}$ are codes and that f and g are morphisms which we do not exclude outright. However, $h = g \circ f$ is a prohibited morphism since β is mapped onto the split vertex α in A; $(1,\beta)$ is the only edge in $\hat{A}$ incident on β; $h(1,\beta) = (1,\alpha) \in e_A \cap d_A^{nv}$; and there is more than one edge in e_A incident on α. Thus, we must exclude at least one of these morphisms as well. However, f is one of the most basic morphisms possible between codes, and g is one of the most elemental morphisms from a code to a non-code precode. We certainly do not wish to exclude such fundamental homomorphisms. We conclude that there is no useful subcategory of the category of

precodes for which we have a factorization theorem analogous to Theorem 7.7 and for which the factorization must always be unique.

7.3. The Relationship Between the Split and Smash.

Recall Definition A.26. We now work toward evidencing a connection between the split of a precode and its smash. Although the statement of the following lemma may seem a bit odd, it will make the proof of the next theorem simpler. We first recall some notation which we will use in the following proofs.

Remark 7.10. *Let $\mathcal{A}$ be a precode. Then*

$$\mathcal{A}^{op} = (C_{\mathcal{A}}, P_{\mathcal{A}}, d_{\mathcal{A}}, e_{\mathcal{A}}),$$

$$(\mathcal{A}^{op})_{\mathcal{W}} = (C_{\mathcal{A}}, C_{(\mathcal{A}^{op})_{\mathcal{W}}}, e_{(\mathcal{A}^{op})_{\mathcal{W}}}, d_{(\mathcal{A}^{op})_{\mathcal{W}}}),$$

and

$$((\mathcal{A}^{op})_{\mathcal{W}})^{op} = (C_{(\mathcal{A}^{op})_{\mathcal{W}}}, C_{\mathcal{A}}, d_{(\mathcal{A}^{op})_{\mathcal{W}}}, e_{(\mathcal{A}^{op})_{\mathcal{W}}}).$$

We also recall that $C_{(\mathcal{A}^{op})_{\mathcal{W}}}$ is the disjoint union $C_{(\mathcal{A}^{op})_{\mathcal{W}}} = \bigcup_{\pi \in P_{\mathcal{A}}} K_{\pi}$ as in Remark 7.4. Finally,

$$(((\mathcal{A}^{op})_{\mathcal{W}})^{op})_{\#} = (C_{(\mathcal{A}^{op})_{\mathcal{W}}}/E, C_{\mathcal{A}}, d_{(\mathcal{A}^{op})_{\mathcal{W}}}/(E,I), e_{(\mathcal{A}^{op})_{\mathcal{W}}}/(I,E)),$$

where E is the smallest equivalence relation containing $e_{(\mathcal{A}^{op})_{\mathcal{W}}} \circ d_{(\mathcal{A}^{op})_{\mathcal{W}}}$.

Lemma 7.11. *Let $\mathcal{A}$ be a **code**, and recall Remark 7.10. If π and ψ are in the same E-equivalence class, then $\pi, \psi \in K_{\alpha}$ for some $\alpha \in P_{\mathcal{A}}$; that is, π and ψ were split from the same element of $P_{\mathcal{A}}$ in the formation of $(\mathcal{A}^{op})_{\mathcal{W}}$.*

Proof. Throughout this proof, let $C = C_{(\mathcal{A}^{op})_{\mathcal{W}}}$, $e = e_{(\mathcal{A}^{op})_{\mathcal{W}}}$, and $d = d_{(\mathcal{A}^{op})_{\mathcal{W}}}$. Suppose π and ψ are in the same E-equivalence class. Since E is the smallest equivalence relation containing $e \circ d$, then there exist distinct $\pi_1, ..., \pi_{n-1} \in C$ such that

$$(\pi_0 = \pi, \pi_1) \in e \circ d \text{ or } (\pi_1, \pi_0 = \pi) \in e \circ d$$

$$(\pi_1, \pi_2) \in e \circ d \text{ or } (\pi_2, \pi_1) \in e \circ d$$

$$\vdots$$

$$(\pi_{n-1}, \pi_n = \psi) \in e \circ d \text{ or } (\pi_n = \psi, \pi_{n-1}) \in e \circ d.$$

Thus, there exist $\kappa_1, \ldots, \kappa_n \in C_{\mathcal{A}}$ such that

$$((\pi_0 = \pi, \kappa_1) \in d \text{ and } (\kappa_1, \pi_1) \in e) \text{ or } ((\pi_1, \kappa_1) \in d \text{ and } (\kappa_1, \pi_0 = \pi) \in e)$$

$$((\pi_1, \kappa_2) \in d \text{ and } (\kappa_2, \pi_2) \in e) \text{ or } ((\pi_2, \kappa_2) \in d \text{ and } (\kappa_2, \pi_1) \in e)$$

$$\vdots$$

$$((\pi_{n-1}, \kappa_n) \in d \text{ and } (\kappa_n, \pi_n = \psi) \in e) \text{ or } ((\pi_n = \psi, \kappa_n) \in d \text{ and } (\kappa_n, \pi_{n-1}) \in e).$$

We now show that for any $0 \leq i \leq n-1$, $\pi_i, \pi_{i+1} \in K_\alpha$ for some $\alpha \in P_{\mathcal{A}}$. Since the K_α are pairwise disjoint, this will show that $\pi, \psi \in K_\alpha$ for some $\alpha \in P_{\mathcal{A}}$. Let $0 \leq i \leq n-1$, and suppose that $\pi_i \in K_\alpha$ and $\pi_{i+1} \in K_\beta$ for some $\alpha, \beta \in P_{\mathcal{A}}$. Now, as above, we have that

$$((\pi_i, \kappa_{i+1}) \in d \text{ and } (\kappa_{i+1}, \pi_{i+1}) \in e) \text{ or } ((\pi_{i+1}, \kappa_{i+1}) \in d \text{ and } (\kappa_{i+1}, \pi_i) \in e).$$

W.l.o.g., suppose that $(\pi_i, \kappa_{i+1}) \in d$ and $(\kappa_{i+1}, \pi_{i+1}) \in e$. Hence, $(\alpha, \kappa_{i+1}) \in e_{\mathcal{A}}$ and $(\kappa_{i+1}, \beta) \in d_{\mathcal{A}}$, so that $(\alpha, \beta) \in d_{\mathcal{A}} \circ e_{\mathcal{A}}$. Since $\mathcal{A}$ is a code, then $\alpha = \beta$. $\qquad\square$

Remark 7.12. *For each $\psi \in C_{(\mathcal{A}^{op})_{M}}$, let E_ψ denote the E-equivalence class of ψ. Then the above lemma shows that if $E_\pi = E_\psi$, then $\pi, \psi \in K_\alpha$ for some $\alpha \in P_{\mathcal{A}}$. In particular each K_π is the disjoint union of some of the E-equivalence classes. That is, the E-equivalence classes form a refinement of the partition $\{K_\alpha \mid a \in P_{\mathcal{A}}\}$ of $P_{\mathcal{A}}$.*

The following lemma is a partial converse of Lemma 7.11.

Lemma 7.13. *Let $\mathcal{A}$ be a **code**, and recall Remark 7.10. Suppose $\pi, \psi \in K_\alpha$ for some $\alpha \in P_\mathcal{A}$, where $\pi \neq \psi$. Then π and ψ are in the same E-equivalence class if and only if $(\alpha, \kappa) \in e_\mathcal{A} \cap d_\mathcal{A}^{nv}$ for some $\kappa \in C_\mathcal{A}$.*

Proof. Throughout this proof, let $C = C_{(\mathcal{A}^{op})_M}$, $e = e_{(\mathcal{A}^{op})_M}$, and $d = d_{(\mathcal{A}^{op})_M}$. Since $|K_\alpha| > 1$, then α is a split vertex with respect to the construction of $(\mathcal{A}^{op})_M$ from $\mathcal{A}^{op}$.

($\Leftarrow$:) Suppose $(\alpha, \kappa) \in e_\mathcal{A} \cap d_\mathcal{A}^{nv}$ for some $\kappa \in C_\mathcal{A}$. Then $(\kappa, \alpha) \in d_\mathcal{A}$ and $(\alpha, \kappa) \in e_\mathcal{A}$. By the definition of $(\mathcal{A}^{op})_M$, $(\kappa, \kappa\alpha\kappa) \in e$ and $(\kappa\alpha\kappa, \kappa) \in d$. We will be done if we can show that π and ψ are each E-equivalent to $\kappa\alpha\kappa$. Since these proofs are similar, we need only show that π and $\kappa\alpha\kappa$ are E-equivalent.

Since $\pi \in K_\alpha$, then there is some edge incident on π in $(\mathcal{A}^{op})_M$. Thus, $(\kappa_\pi, \pi) \in e$ or $(\pi, \kappa_\pi) \in d$ for some $\kappa_\pi \in C_\mathcal{A}$. W.l.o.g., suppose that $(\pi, \kappa_\pi) \in d$. Thus, either $\pi = \alpha_{out}$ or $\pi = \delta\alpha\delta$ for some $\delta \in C_\mathcal{A}$.

Case 1: Suppose that $\pi = \delta\alpha\delta$ for some $\delta \in C_\mathcal{A}$. Then $(\delta, \delta\alpha\delta) \in e$ and $(\delta\alpha\delta, \delta) \in d$. By Algorithm 7.2, we note that $\delta\alpha\delta \in K_\alpha$ if and only if $_{in}\alpha \in K_\alpha$ or $\alpha_{out} \in K_\alpha$. W.l.o.g., suppose $_{in}\alpha \in K_\alpha$. Thus, $(\delta, _{in}\alpha) \in e$. Also, since $(\kappa, \alpha) \in d_\mathcal{A}$, then $(\kappa, _{in}\alpha) \in e$. But, $(\kappa\alpha\kappa, \kappa) \in d$ and $(\kappa, _{in}\alpha) \in e$ imply that $\kappa\alpha\kappa$ and $_{in}\alpha$ are in the same E-equivalence class. Similarly, $(\delta\alpha\delta, \delta) \in d$ and $(\delta, _{in}\alpha) \in e$ imply that $\pi = \delta\alpha\delta$ and $_{in}\alpha$ are in the same E-equivalence class. Hence, $\kappa\alpha\kappa$ and π are E-equivalent.

Case 2: Suppose that $\pi = \alpha_{out}$. Then $(\alpha_{out}, \kappa) \in d$. Since $(\pi = \alpha_{out}, \kappa) \in d$ and $(\kappa, \kappa\alpha\kappa) \in e$, then π and $\kappa\alpha\kappa$ are E-equivalent.

($\Rightarrow$:) Suppose π and ψ are in the same E-equivalence class, and assume there is no $\kappa \in C_\mathcal{A}$ such that $(\alpha, \kappa) \in e_\mathcal{A} \cap d_\mathcal{A}^{nv}$. Thus, there are no elements of the form $\kappa\alpha\kappa$ in K_α. Since $\pi \neq \psi$, then $\{\pi, \psi\} = K_\alpha = \{_{in}\alpha, \alpha_{out}\}$. W.l.o.g., suppose $\pi =_{in} \alpha$ and $\psi = \alpha_{out}$.

Since E is the smallest equivalence relation containing $e \circ d$, then π and ψ being in the same E-class means there exist distinct $\pi_1, ..., \pi_{n-1} \in C$ such that

$$(\pi, \pi_1) \in e \circ d \text{ or } (\pi_1, \pi) \in e \circ d$$

$$(\pi_1, \pi_2) \in e \circ d \text{ or } (\pi_2, \pi_1) \in e \circ d$$

$$\vdots$$

$$(\pi_{n-1}, \psi) \in e \circ d \text{ or } (\psi, \pi_{n-1}) \in e \circ d.$$

Thus, there exist $\kappa_1, ..., \kappa_n \in C_{\mathcal{A}}$ such that

$$((\pi, \kappa_1) \in d \text{ and } (\kappa_1, \pi_1) \in e) \text{ or } ((\pi_1, \kappa_1) \in d \text{ and } (\kappa_1, \pi) \in e)$$

$$((\pi_1, \kappa_2) \in d \text{ and } (\kappa_2, \pi_2) \in e) \text{ or } ((\pi_2, \kappa_2) \in d \text{ and } (\kappa_2, \pi_1) \in e)$$

$$\vdots$$

$$((\pi_{n-1}, \kappa_n) \in d \text{ and } (\kappa_n, \psi) \in e) \text{ or } ((\psi, \kappa_n) \in d \text{ and } (\kappa_n, \pi_{n-1}) \in e).$$

W.l.o.g., suppose that $(\pi, \kappa_1) \in d$ and $(\kappa_1, \pi_1) \in e$. Now, $\pi_1 \in K_{\alpha_1}$ for some $\alpha_1 \in P_{\mathcal{A}}$. Since $\pi \in K_\alpha$, then $(\alpha, \kappa_1) \in e_{\mathcal{A}}$ and $(\kappa_1, \alpha_1) \in d_{\mathcal{A}}$, so that $(\alpha, \alpha_1) \in d_{\mathcal{A}} \circ e_{\mathcal{A}}$. Since $\mathcal{A}$ is a code, then $\alpha = \alpha_1$. Thus, $(\alpha, \kappa_1) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$. However, this contradicts that there is no $\kappa \in C_{\mathcal{A}}$ such that $(\alpha, \kappa) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$. $\qquad\square$

Remark 7.14. *The proof of the above lemma shows that for $\alpha \in P_{\mathcal{A}}$ such that $|K_\alpha| > 1$, there are two cases:*

1) $K_\alpha = \{_{in}\alpha, \alpha_{out}\}$, in which case $\{_{in}\alpha\}$ and $\{\alpha_{out}\}$ are distinct E-equivalence classes.

2) K_α contains an element of the form $\kappa\alpha\kappa$ for some $\kappa \in C_{\mathcal{A}}$, in which case K_α is contained in an E-equivalence class. By Lemma 7.11, then K_α must be an E-equivalence class.

Lemmas 7.11 and 7.13 (and their proofs) give us the following relationship between the codes $\mathcal{A}$ and $((((\mathcal{A}^{op})_{\mathsf{M}})^{op})_{\#})$.

Theorem 7.15. *Let $\mathcal{A}$ be a code. Then the structure of $((((\mathcal{A}^{op})_{\mathsf{M}})^{op})_{\#})$ is determined locally for each $\pi \in P_{\mathcal{A}}$ as follows:*

If either $K_\pi = \{\pi\}$ or $K_\pi \neq \{\pi\}$ and there is some $\kappa \in C_A$ such that $(\pi, \kappa) \in e_A \cap d_A^{nv}$, then the E-equivalence class containing π is K_π. Thus, $(\pi, k) \in e_A$ if and only if $(K_\pi, k) \in d_{(A^{op})_M}/(E, I)$, and $(k, \pi) \in d_A$ if and only if $(k, K_\pi) \in e_{(A^{op})_M}/(I, E)$.

If $K_\pi \neq \{\pi\}$ and there is no $(\pi, \kappa) \in e_A \cap d_A^{nv}$ for any $\kappa \in C_A$, then $K_\pi = \{_{in}\pi, \pi_{out}\}$. In this case, there are two E-classes associated with K_π: $\{_{in}\pi\}$ and $\{\pi_{out}\}$. Moreover, $(\pi, k) \in e_A$ if and only if $(\{\pi_{out}\}, k) \in d_{(A^{op})_M}/(E, I)$, and $(k, \pi) \in d_A$ if and only if $(k, \{_{in}\pi\}) \in e_{(A^{op})_M}/(I, E)$.

Remark 7.16. *The above theorem simply says that if A is a code, then $((((A^{op})_M)^{op})_\#) = \mathcal{B} = (P_\mathcal{B}, C_\mathcal{B}, e_\mathcal{B}, d_\mathcal{B})$ is the minimal code having A as a homomorphic image and which satisfies the following condition:*

If $\pi \in P_\mathcal{B}$ and $(\pi, \pi) \notin d_\mathcal{B} \circ e_\mathcal{B}$, then there are no $\kappa, \delta \in C_\mathcal{B}$ such that $(\pi, \kappa) \in e_\mathcal{B}$ and $(\delta, \pi) \in d_\mathcal{B}$. This condition is equivalent to the following one:

$$c_\mathcal{B} VD = e_\mathcal{B} VD \cup d_\mathcal{B} VD, \text{ where}$$

$$e_\mathcal{B} VD = P_\mathcal{B} \backslash DOM(e_\mathcal{B}), d_\mathcal{B} VD = P_\mathcal{B} \backslash RAN(d_\mathcal{B}), \text{ and } c_\mathcal{B} VD = P_\mathcal{B} \backslash DOM(c_\mathcal{B})$$

as in Definition A.11.

Note that if we view codes as graphs, then the above characterization implies that $((((A^{op})_M)^{op})_\#)$ is the direct sum (see Definition A.25) of the connected components of A with plaintext vertices added as necessary to satisfy the above condition.

Example 7.17. *Consider $A = (P_A = \{p_1, p_2\}, C_A = \{c_1, c_2, c_3\}, e_A, d_A)$, where*

$$e_A = \{(p_1, c_2), (p_1, c_1), (p_2, c_2)\} \text{ and } d_A = \{(c_1, p_1), (c_3, p_1), (c_3, p_2))\}.$$

Then $\mathcal{B} = (((A^{op})_M)^{op})_\# = (C_{(A^{op})_M}/E, C_A, d_{(A^{op})_M}/(E, I), e_{(A^{op})_M}/(I, E))$, where $C_{(A^{op})_M}/E = \{K_{p_1}, \{p_2\}, \{p_3\}\}$, $d_{(A^{op})_M}/(E, I) = \{(K_{p_1}, c_1), (K_{p_1}, c_2), (\{p_2\}, c_2)\}$, and

$$e_{(A^{op})_M}/(I, E) = \{(c_1, K_{p_1}), (c_3, K_{p_1}), (c_3, \{p_3\})\}.$$

The plots of $\mathcal{A}$ and $\mathcal{B}$ in Figure 10 were generated using the Maple code given in Appendix C. We note that the program represents elements of the form α_{out} via α_out, etc. Also, when it generates the smash of a precode, it chooses a representative for an equivalence class to represent the class. Thus, for example, it represents K_{p_1} with $p1\_out$.
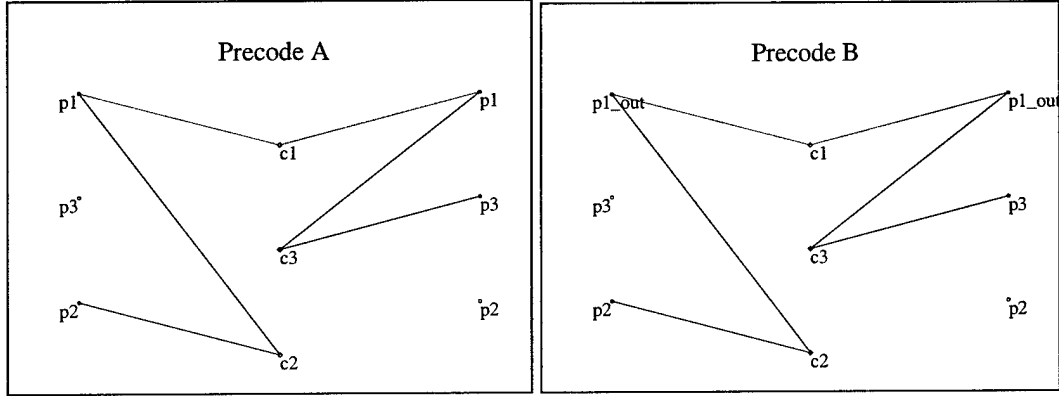
FIGURE 10. $\mathcal{A}$ Is Isomorphic to $\mathcal{B} = (((\mathcal{A}^{op})_{\mathsf{M}})^{op})_{\#}$

Example 7.18. *Consider* $\mathcal{A} = (P_\mathcal{A} = \{p_1, p_2, p_3\}, C_\mathcal{A} = \{c_1, c_2\}, e_\mathcal{A}, d_\mathcal{A})$, *where*

$$e_\mathcal{A} = \{(p_1, c_1), (p_2, c_1)\} \ and \ d_\mathcal{A} = \{(c_2, p_1), (c_2, p_3)\}.$$

Then $\mathcal{B} = (((\mathcal{A}^{op})_{\mathsf{M}})^{op})_{\#} = (C_{(\mathcal{A}^{op})_{\mathsf{M}}}/E, C_\mathcal{A}, d_{(\mathcal{A}^{op})_{\mathsf{M}}}/(E, I), e_{(\mathcal{A}^{op})_{\mathsf{M}}}/(I, E))$, *where*

$C_{(\mathcal{A}^{op})_{\mathsf{M}}}/E = \{\{_{in}p_1\}, \{_{in}p_2\}, \{p_{1\,out}\}, \{p_{2\,out}\}\}$, $d_{(\mathcal{A}^{op})_{\mathsf{M}}}/(E, I) = \{(\{p_{1\,out}\}, c_1), (\{p_{2\,out}\}, c_1)\}$, *and*

$$e_{(\mathcal{A}^{op})_{\mathsf{M}}}/(I, E) = \{(c_2, \{_{in}p_1\}), (c_2, \{_{in}p_2\})\}.$$

See Figure 11.
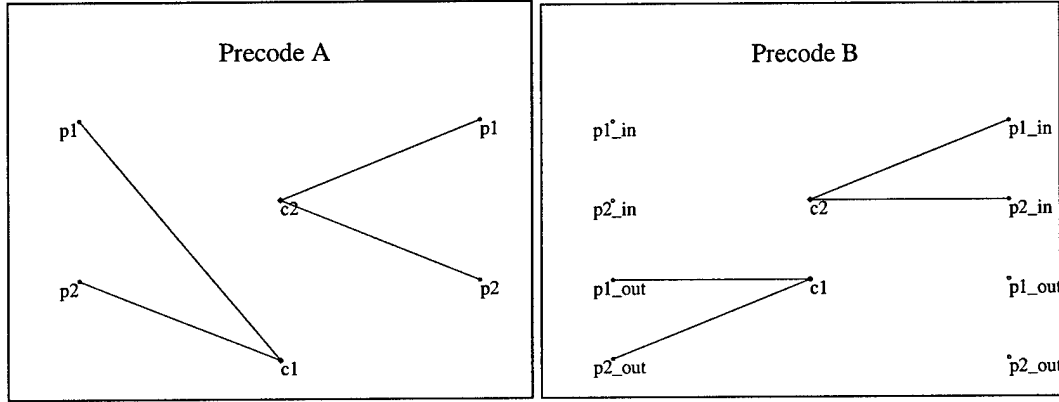
FIGURE 11. $\mathcal{A}$ Is Not Isomorphic to $\mathcal{B} = (((\mathcal{A}^{op})_{\mathsf{N}})^{op})_{\#}$

7.4. The Bald-Split of a Precode. The following "split" is a subprecode of the one given by Algorithm 7.2.

Algorithm 7.19. *Let $\mathcal{A}$ be a precode. We construct a code $\mathcal{A}_{||}$, called the **bald-split of** $\mathcal{A}$, as follows:*

Set $P_{\mathcal{A}_{||}} = P_{\mathcal{A}}$, $C_{\mathcal{A}_{||}} = \emptyset$, $e_{\mathcal{A}_{||}} = \emptyset$, and $d_{\mathcal{A}_{||}} = \emptyset$.

For each $\kappa \in C_{\mathcal{A}}$

   If $\exists \pi_1, \pi_2 \in P_{\mathcal{A}}$ such that $\pi_1 \neq \pi_2, (\pi_1, \kappa) \in e_{\mathcal{A}}$, and $(\pi_2, \kappa) \in d_{\mathcal{A}}^{nv}$, then

      /* κ must be split */

      For each $\pi \in P_{\mathcal{A}}$ such that $(\pi, \kappa) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv}$

         Add $\pi\kappa\pi$ to $C_{\mathcal{A}_{||}}$

         Add $(\pi, \pi\kappa\pi)$ to $e_{\mathcal{A}_{||}}$

         Add $(\pi, \pi\kappa\pi)$ to $d_{\mathcal{A}_{||}}^{nv}$

      end for

(*)   *If $\exists \pi_1 \in P_{\mathcal{A}}$ such that $(\pi_1, \kappa) \in e_{\mathcal{A}} \backslash d_{\mathcal{A}}^{nv}$, then*

      Add $_{in}\kappa$ to $C_{\mathcal{A}_{||}}$

(*)   *For each $\pi \in P_{\mathcal{A}}$ such that $(\pi, \kappa) \in e_{\mathcal{A}} \backslash d_{\mathcal{A}}^{nv}$*

$$Add\ (\pi,_{in} \kappa)\ to\ e_{\mathcal{A}_{||}}$$

$$end\ for$$

$$end\ if$$

(*)   *If* $\exists \pi_1 \in P_{\mathcal{A}}$ *such that* $(\pi_1, \kappa) \in d_{\mathcal{A}}^{nv} \backslash e_{\mathcal{A}}$, *then*

$$Add\ \kappa_{out}\ to\ C_{\mathcal{A}_{||}}$$

(*)   *For each* $\pi \in P_{\mathcal{A}}$ *such that* $(\pi, \kappa) \in d_{\mathcal{A}}^{nv} \backslash e_{\mathcal{A}}$

$$Add\ (\pi, \kappa_{out})\ to\ d_{\mathcal{A}_{||}}^{nv}$$

$$end\ for$$

$$end\ if$$

else

/* κ *does not need to be split* */

$Add\ \kappa\ to\ C_{\mathcal{A}_{||}}$

For each $\pi \in P_{\mathcal{A}}$

If $(\pi, \kappa) \in e_{\mathcal{A}}$, *then*

$$Add\ (\pi, \kappa)\ to\ e_{\mathcal{A}_{||}}$$

$$end\ if$$

If $(\pi, \kappa) \in d_{\mathcal{A}}^{nv}$, *then*

$$Add\ (\pi, \kappa)\ to\ d_{\mathcal{A}_{||}}^{nv}$$

$$end\ if$$

$$end\ for$$

$$end\ if$$

$$end\ for$$

We can give an alternate description of Algorithm 7.19 using the synoptic codebook matrices $M_{\mathcal{A}}$ and $M_{\mathcal{A}_{||}}$. Let $M_{\mathcal{A}}(\kappa)(\pi)$ denote $M_{\mathcal{A}}(\pi, \kappa)$, the π-th entry of the κ-th column of $M_{\mathcal{A}}$. In

86

the following algorithm, we use a question mark as a wildcard character. That is, we use 1? to

represent either symbol in $\{e = 10, s = 11\}$ and ?1 to represent either symbol in $\{d = 01, s = 11\}$.

Algorithm 7.20. *Let $\mathcal{A}$ be a precode. We construct $M_{A_{||}}$ from M_A as follows:*

Set $P_{A_{||}} = P_A$, $C_{A_{||}} = \emptyset$, $e_{A_{||}} = \emptyset$, and $d_{A_{||}} = \emptyset$.

For each κ in C_A

    If $M_A(\kappa)(\pi) = 1?$ and $M_A(\kappa)(\psi) = ?1$ for some $\pi, \psi \in P_A$ such that $\pi \neq \psi$, then

        /* κ must be split */

        For each $\psi \in P_A$ such that $M_A(\kappa)(\psi) = 11$

            Add a column to $M_{A_{||}}$ with a 11 in row ψ and 00's elsewhere.

        end for

        If $M_A(\kappa)(\pi) = 10$ for some $\pi \in P_A$, then

            Add a column to $M_{A_{||}}$ with a 10 in each row

              for which there is a 10 in $M_A(\kappa)$ and with 00's elsewhere

        end if

        If $M_A(\kappa)(\pi) = 01$ for some $\pi \in P_A$, then

            Add a column to $M_{A_{||}}$ with a 01 in each row

              for which there is a 01 in $M_A(\kappa)$ and with 00's elsewhere

        end if

    else

        /* κ does not need to be split */

        Add a column to $M_{A_{||}}$ identical to $M_A(\kappa)$

    end if

end for

Remark 7.21. *In constructing $\mathcal{A}_{||}$, for each $\kappa \in C_A$, we add to $C_{A_{||}}$ some nonempty subset $K_\kappa \subseteq$*

$(\{_{in}\kappa, \kappa_{out}, \kappa\} \cup \bigcup_{\pi \in P_A} \{\pi \kappa \pi\})$. Furthermore, $P_{A_{||}} = P_A$, and there is a one-to-one correspondence

between e_A and $e_{A_{||}}$ such that $(\pi, \kappa) \in e_A$ if and only if $(\pi, \chi) \in e_{A_{||}}$ for some $\chi \in K_\kappa$. Similarly, there is a one-to-one correspondence between d_A and $d_{A_{||}}$ such that $(\kappa, \pi) \in d_A$ if and only if $(\chi, \pi) \in d_{A_{||}}$ for some $\chi \in K_\kappa$. Moreover, there is a one-to-one correspondence between $e_A \cap d_A^{nv}$ and $e_{A_{||}} \cap d_{A_{||}}^{nv}$ such that $(\pi, \kappa) \in e_A \cap d_A^{nv}$ if and only if $(\pi, \chi) \in e_{A_{||}} \cap d_{A_{||}}^{nv}$ for some $\chi \in K_\kappa$. Hence, there is a canonical strong epimorphism $k = (k_1, k_2) : A_{||} \to A$ such that $k_1 = 1_{P_A}$ and for each $\kappa \in C_A$, $k_2(K_\kappa) = \{\kappa\}$. We say that $\kappa \in C_A$ is a **split vertex** if $|K_\kappa = k_2^{-1}(\kappa)| > 1$.

We note that Algorithm 7.19 differs from Algorithm 7.2 only on the lines marked with (*).

The following theorem follows from the above remarks.

Theorem 7.22. *Let A be a precode. Then $A_{||} \neq A_{\mathcal{M}}$ if and only if there exists a split vertex $\kappa \in C_A$ such that at least one of the following two conditions holds:*

    (i) There exist $\pi_1, \pi_2 \in P_A$ such that $\pi_1 \neq \pi_2, (\pi_1, \kappa) \in e_A$, and $(\pi_2, \kappa) \in e_A \backslash d_A^{nv}$

    (ii) There exist $\pi_1, \pi_2 \in P_A$ such that $\pi_1 \neq \pi_2, (\pi_1, \kappa) \in d_A^{nv}$, and $(\pi_2, \kappa) \in d_A^{nv} \backslash e_A$.

Theorem 7.23. *Let A be a precode with bald-split $A_{||} = (P_{A_{||}}, C_{A_{||}}, e_{A_{||}}, d_{A_{||}})$ and canonical strong epimorphism $k = (k_1, k_2) : A_{||} \to A$. Let $\hat{A}$ be a **code** with no isolated codetext elements and $h = (h_1, h_2) : \hat{A} \to A$ be a precode homomorphism. If there exists a precode homomorphism $f = (f_1, f_2) : \hat{A} \to A_{||}$ making the diagram*

$$
\begin{array}{ccc}
A & \xleftarrow{\;\;k\;\;} & A_{||} \\
{\scriptstyle h}\big\uparrow & \nearrow & \\
 & {\scriptstyle f} & \\
\hat{A} & &
\end{array}
$$

commute (i.e. $h = k \circ f$), then f is unique.

Proof. Suppose that $f = (f_1, f_2) : \hat{A} \to A_{||}$ is a precode homomorphism satisfying $h = k \circ f$. By definition, $P_{A_{||}} = P_A$ and $k_1 = 1_{P_A}$. Since $k_1 \circ f_1 = h_1$, then $f_1 = h_1$. Hence, f_1 is unique.

Let $\hat{\kappa} \in C_{\hat{A}}$ and $\kappa = h_2(\hat{\kappa})$. Since $\hat{A}$ is a code and $\hat{\kappa}$ is not isolated, then there exists $\hat{\pi} \in P_{\hat{A}}$ such that $(\hat{\pi}, \hat{\kappa}) \in e_{\hat{A}}$ or $(\hat{\kappa}, \hat{\pi}) \in d_{\hat{A}}$. W.l.o.g., suppose that $(\hat{\pi}, \hat{\kappa}) \in e_{\hat{A}}$, and let $\pi = h_1(\hat{\pi})$. Since h is a homomorphism, then $(\pi, \kappa) \in e_A$. As in Remark 7.21, there is a unique $\chi \in P_{A_{||}}$ such that $(\pi, \chi) \in e_{A_{||}}$. Thus, we must have $f_2(\hat{\kappa}) = \chi$, and f_2 is unique. $\qquad \square$

Recall that Theorem 7.7 shows that we can factor any homomorphism from a code $\hat{A}$ to a precode A through $A_{\backslash\!/}$. We immediately have the following corollary to Theorem 7.23.

Corollary 7.24. *Let A be a precode such that $A_{||} = A_{\backslash\!/}$. Let $\hat{A}$ be a **code** with no isolated codetext elements and $h = (h_1, h_2) : \hat{A} \to A$ be a precode homomorphism. Let $k = (k_1, k_2) : A_{||} \to A$ be the canonical strong epimorphism. Then there exists a unique precode homomorphism $f = (f_1, f_2) : \hat{A} \to A_{||}$ such that $h = k \circ f$.*

We note that Corollary 7.24 and Theorem 7.22 give conditions on the structure of A which guarantee the existence and uniqueness of a factorization through $A_{||}$. We now give conditions on the code $\hat{A}$ which guarantee the existence and uniqueness of the factorization through $A_{||}$.

Definition 7.25. *Let $A = (P_A \neq \emptyset, C_A \neq \emptyset, e_A, d_A)$ be a precode.*

For each $\kappa \in C_A$, let $P_\kappa = \{\pi \in P_A \mid (\pi, \kappa) \in e_A \cup d_A^{nv}\}$ and set $n_\kappa = |P_\kappa|$. Let n be the supremum of $\{n_\kappa \mid \kappa \in C_A\}$. Then A is said to be of C-order n, and we call A a C_n precode. Note that n may be infinite.

Remark 7.26. *For $n \in \mathbb{N}$, each codetext element in a C_n precode is adjacent to no more than n plaintext elements. Furthermore, the C_1 **codes** are precisely those for which each codetext element of type e or d has precisely one edge incident on it.*

Theorem 7.27. *Let A be a precode with bald-split $A_{||}$ and canonical strong epimorphism $k = (k_1, k_2) : A_{||} \to A$. Let $\hat{A}$ be a C_1 **code** and $h = (h_1, h_2) : \hat{A} \to A$ be a precode homomorphism. Then there exists a precode homomorphism $f = (f_1, f_2) : \hat{A} \to A_{||}$ making the diagram*

$$\mathcal{A} \xleftarrow{\quad k \quad} \mathcal{A}_{||}$$

(with h going up from $\hat{\mathcal{A}}$ to $\mathcal{A}$, and f going diagonally from $\hat{\mathcal{A}}$ to $\mathcal{A}_{||}$)

$$\hat{\mathcal{A}}$$

commute; that is, $h = k \circ f$. Furthermore, if $\hat{\mathcal{A}}$ has no isolated codetext elements, then f is unique.

Proof. We note that if $\hat{\mathcal{A}}$ has no isolated codetext elements, then by Theorem 7.23, f must be unique if it exists. By definition, $P_{\mathcal{A}_{||}} = P_{\mathcal{A}}$ and $k_1 = 1_{P_{\mathcal{A}}}$. Since we need $k_1 \circ f_1 = h_1$, then we must set $f_1 = h_1$.

Let $\hat{\kappa} \in C_{\hat{\mathcal{A}}}$ and $\kappa = h_2(\hat{\kappa})$. We now show how to define $f_2(\hat{\kappa})$. If $\hat{\kappa}$ is isolated, we may choose any $\chi \in k_2^{-1}(h_2(\hat{\kappa}))$ and set $f_2(\hat{\kappa}) = \chi$. If $\hat{\kappa}$ is not isolated, then since $\hat{\mathcal{A}}$ is a C_1 code, there is precisely one $\hat{\pi} \in P_{\hat{\mathcal{A}}}$ for which $(\hat{\pi}, \hat{\kappa}) \in e_{\hat{\mathcal{A}}} \cup d_{\hat{\mathcal{A}}}^{nv}$. Let $\pi = h_1(\hat{\pi})$. Since h is a homomorphism, then $(\pi, \kappa) \in e_{\mathcal{A}}$ if $(\hat{\pi}, \hat{\kappa}) \in e_{\hat{\mathcal{A}}}$ and $(\pi, \kappa) \in d_{\mathcal{A}}^{nv}$ if $(\hat{\pi}, \hat{\kappa}) \in d_{\hat{\mathcal{A}}}^{nv}$. Furthermore, by Remark 7.21, we know that there is some $\chi \in K_{\kappa}$ such that $(\pi, \chi) \in e_{\mathcal{A}_{||}}$ if $(\pi, \kappa) \in e_{\mathcal{A}}$ and $(\chi, \pi) \in d_{\mathcal{A}_{||}}$ if $(\kappa, \pi) \in d_{\mathcal{A}}$. Thus, we may set $f_2(\hat{\kappa}) = \chi$. $\square$

Definition 7.28. *Let* $\mathcal{A} = (P_{\mathcal{A}} \neq \emptyset, C_{\mathcal{A}} \neq \emptyset, e_{\mathcal{A}}, d_{\mathcal{A}})$ *be a precode such that* $e_{\mathcal{A}}$ *and* $d_{\mathcal{A}}^{nv}$ *are surjective relations; that is, for each* $\kappa \in C_{\mathcal{A}}$, *there exist* $(\pi, \kappa) \in e_{\mathcal{A}}$ *and* $(\psi, \kappa) \in d_{\mathcal{A}}^{nv}$ *for some* $\pi, \psi \in P_{\mathcal{A}}$. *Then* $\mathcal{A}$ *is said to be a* C-full *precode. If* $\mathcal{A}$ *is a code, we call it a* C-full *code, and we note that* $\mathcal{A}$ *is self-companion.*

Lemma 7.29. *Suppose that* $\mathcal{A}$ *is a* C-full *precode with bald-split* $\mathcal{A}_{||}$. *Then* $\mathcal{A}_{||}$ *is a* C-full *code (and is therefore self-companion) if and only if* $\mathcal{A}$ *is a self-companion precode.*

Proof. We have the canonical strong epimorphism $k = (k_1, k_2) : \mathcal{A}_{||} \to \mathcal{A}$.

($\Leftarrow$:) Suppose $\mathcal{A}$ is self-companion. Then $e_{\mathcal{A}} = e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv} = d_{\mathcal{A}}^{nv}$. Let $\kappa \in C_{\mathcal{A}}$, and let $P_{\mathcal{A}}' = \{\pi \in P_{\mathcal{A}} | (\pi, \kappa) \in e_{\mathcal{A}}\}$. Since $\mathcal{A}$ is C-full, then $|P_{\mathcal{A}}'| \neq 0$. If $|P_{\mathcal{A}}'| > 1$, then as in Algorithm 7.19, $K_{\kappa} = k_2^{-1}(\kappa) = \bigcup_{\pi \in P_{\mathcal{A}}'} \pi \kappa \pi$. Furthermore, $(\pi, \pi \kappa \pi) \in e_{\mathcal{A}_{||}} \cap d_{\mathcal{A}_{||}}^{nv}$ for each $\pi \kappa \pi \in K_{\kappa}$. If $|P_{\mathcal{A}}'| = 1$,

then $P'_A = \{\pi\}$ for some $\pi \in P_A$. As in Algorithm 7.19, then $K_\kappa = \{\kappa\}$ and $(\pi, \kappa) \in e_{A_{||}} \cap d^{nv}_{A_{||}}$. Since κ was chosen arbitrarily, then $e_{A_{||}}$ and $d^{nv}_{A_{||}}$ are surjective.

($\Rightarrow$:) Suppose $A_{||}$ is a C-full code. Since k is a strong precode homomorphism and $A_{||}$ is self-companion, then by Theorem 8B4 in [3], $A = k(A_{||})$ is self-companion. □

Definition 7.30. *We call a self-companion C-full precode an SCF-precode. We note that the SCF-precodes are precisely the self-companion precodes with no isolated codetext elements. The SCF-codes are the self-companion precodes with no isolated codetext elements; that is, they are the strictly S-codes as defined in Definition 4.2. We also note that the SCF-codes are C_1 codes as defined in Definition 7.25.*

Let $\mathfrak{P}_{SCF}$ be the category of SCF-precodes and precode homomorphisms. The following corollary of Theorem 7.27 is the analogue of Theorem 7.7 for the category $\mathfrak{P}_{SCF}$.

Corollary 7.31. *Let A be an SCF-precode with associated bald-split code $A_{||}$ and canonical strong epimorphism $k : A_{||} \to A$. Let $\hat{A}$ be an SCF-code and $h : \hat{A} \to A$ be a precode homomorphism. Then there exists a unique precode homomorphism $f : \hat{A} \to A_{||}$ such that $h = k \circ f$.*

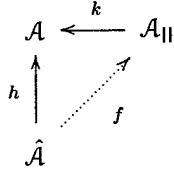The next example demonstrates the relationships evidenced in this section.

Example 7.32. *Let $P_A = P_{A_1} = P_{A_2} = P_{A_3} = \{1,2\}$ and $C_A = C_{A_1} = C_{A_2} = C_{A_3} = \{a\}$.*

Let $A = (P_A, C_A, e_A, d_A)$, where $e_A = \{(1,a),(2,a)\}$ and $d_A = \{(a,1),(a,2)\}$.

Let $A_1 = (P_{A_1}, C_{A_1}, e_{A_1}, d_{A_1})$, where $e_{A_1} = \{(1,a),(2,a)\}$ and $d_{A_1} = \emptyset$.

Let $A_2 = (P_{A_2}, C_{A_2}, e_{A_2}, d_{A_2})$, where $e_{A_2} = \{(1,a)\}$ and $d_{A_2} = \emptyset$.

Let $A_3 = (P_{A_3}, C_{A_3}, e_{A_3}, d_{A_3})$, where $e_{A_3} = \{(1,a)\}$ and $d_{A_3} = \{(a,1)\}$.

See Figure 12 for plots of these precodes.
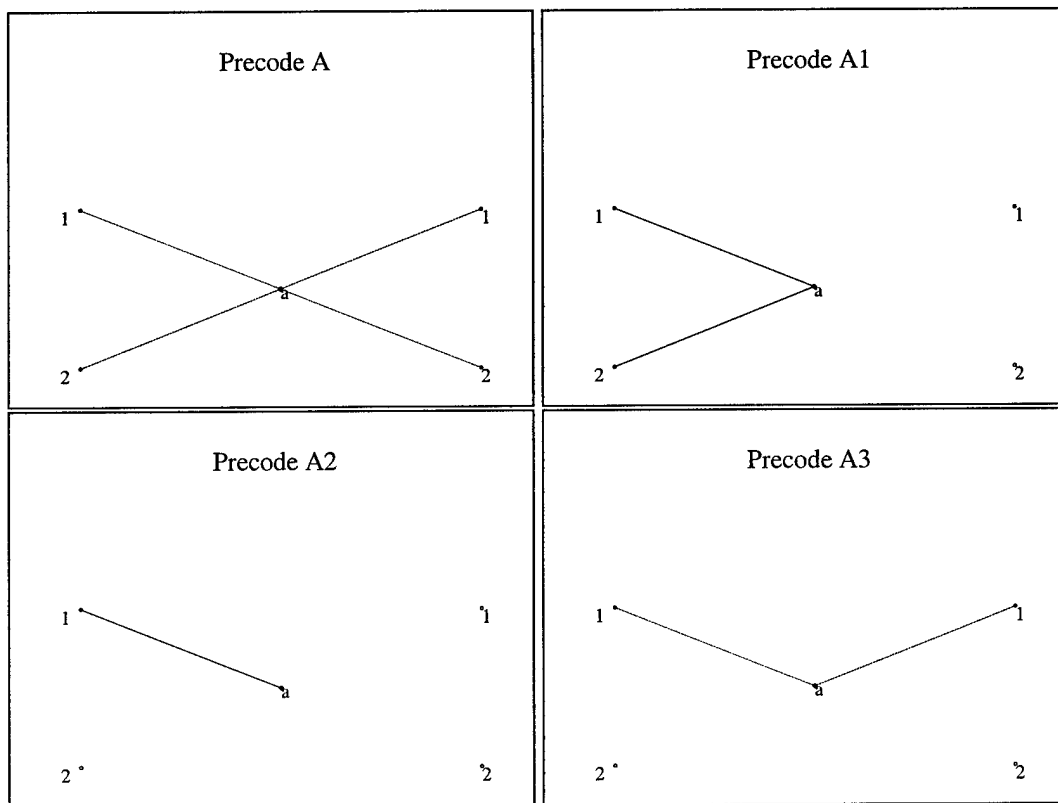
FIGURE 12. The Precodes $\mathcal{A}$, $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$

Note that $\mathcal{A}$ is a self-companion precode. Furthermore, $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ are codes. $\mathcal{A}_1$ is not a

C_1 code and is therefore also not self-companion. $\mathcal{A}_2$ is a C_1 code, but it is not self-companion.

Finally, $\mathcal{A}_3$ is a self-companion code.

Note that $h = 1_A$ can be viewed as a precode homomorphism from each of $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ to $\mathcal{A}$.

Furthermore,

1) $h : \mathcal{A}_1 \longrightarrow \mathcal{A}$ does not factor through $\mathcal{A}_{||}$, but it factors uniquely through $\mathcal{A}_{|\wedge|}$.

2) $h : \mathcal{A}_2 \longrightarrow \mathcal{A}$ factors through both $\mathcal{A}_{||}$ and $\mathcal{A}_{|\wedge|}$. The factorization through $\mathcal{A}_{||}$ is unique; the

factorization through $\mathcal{A}_{|\wedge|}$ is not.

3) $h : \mathcal{A}_3 \longrightarrow \mathcal{A}$ factors uniquely through both $\mathcal{A}_{||}$ and $\mathcal{A}_{|\wedge|}$.

8. PARAMETRIZATION

In calculus, it is sometimes possible to represent a curve in $\mathbb{R}^2$ given by a relation in the variables x and y via functions $f, g : T \longrightarrow \mathbb{R}$, where $T \subseteq \mathbb{R}$. The functions are called parametric equations and together they form a parametrization of the curve. For example, the unit circle is given by the relation $x^2 + y^2 = 1$. One possible parametrization of this curve is defined by setting $T = (\frac{\pi}{2}, \frac{3\pi}{2})$ and by defining $f(t) = cos(t)$ and $g(t) = sin(t)$. The unit circle consists of all the points $(x = f(t), y = g(t))$ where t ranges over T. We now develop an analogue of parametrization for precodes.

Definition 8.1. *Let $\mathcal{A}$ be a precode, and let T be a set. A pair of codes $(\mathcal{F}, \mathcal{G})$ is called a parametrization of $\mathcal{A}$ in T if $P_{\mathcal{F}} = P_{\mathcal{A}}$, $P_{\mathcal{G}} = C_{\mathcal{A}}$, $C_{\mathcal{F}} = C_{\mathcal{G}} = T$, $d_{\mathcal{G}} \circ e_{\mathcal{F}} = e_{\mathcal{A}}$, and $d_{\mathcal{F}} \circ e_{\mathcal{G}} = d_{\mathcal{A}}$. Since $e_{\mathcal{F}} \subseteq P_{\mathcal{A}} \times T$, $d_{\mathcal{F}} \subseteq T \times P_{\mathcal{A}}$, $e_{\mathcal{G}} \subseteq C_{\mathcal{A}} \times T$, $d_{\mathcal{G}} \subseteq T \times C_{\mathcal{A}}$, $e_{\mathcal{A}} \subseteq P_{\mathcal{A}} \times C_{\mathcal{A}}$, and $d_{\mathcal{A}} \subseteq C_{\mathcal{A}} \times P_{\mathcal{A}}$, then $d_{\mathcal{G}} \circ e_{\mathcal{F}} \subseteq P_{\mathcal{A}} \times C_{\mathcal{A}}$ and $d_{\mathcal{F}} \circ e_{\mathcal{G}} \subseteq C_{\mathcal{A}} \times P_{\mathcal{A}}$. Thus, the requirement that $d_{\mathcal{G}} \circ e_{\mathcal{F}} = e_{\mathcal{A}}$ and $d_{\mathcal{F}} \circ e_{\mathcal{G}} = d_{\mathcal{A}}$ makes sense.*

8.1. A First Parametrization. We start with an "obvious" parametrization.

Algorithm 8.2. *Let $\mathcal{A}$ be a precode. We can construct a parametrization $(\mathcal{F}, \mathcal{G})$ for $\mathcal{A}$ as follows:*

Set $T = P_{\mathcal{A}} \times C_{\mathcal{A}}$.

Set $P_{\mathcal{F}} = P_{\mathcal{A}}$, $P_{\mathcal{G}} = C_{\mathcal{A}}$, $C_{\mathcal{F}} = T$, and $C_{\mathcal{G}} = T$.

Set $e_{\mathcal{F}} = \emptyset$, $d_{\mathcal{F}} = \emptyset$, $e_{\mathcal{G}} = \emptyset$, and $d_{\mathcal{G}} = \emptyset$.

For each $(\pi, \kappa) \in e_{\mathcal{A}}$

    Add $(\pi, (\pi, \kappa))$ to $e_{\mathcal{F}}$

    Add $(\kappa, (\pi, \kappa))$ to $d_{\mathcal{G}}^{nv}$

end for

For each $(\kappa, \pi) \in d_{\mathcal{A}}$

    Add $(\pi, (\pi, \kappa))$ to $d_{\mathcal{F}}^{nv}$

Add $(\kappa, (\pi, \kappa))$ *to* $e_{\mathcal{G}}$

end for

Theorem 8.3. *Let* $\mathcal{A}$ *be a precode, and let* $\mathcal{F}$ *and* $\mathcal{G}$ *be as defined in Algorithm 8.2. Then* $(\mathcal{F}, \mathcal{G})$ *is a parametrization of* $\mathcal{A}$ *in* $T = P_{\mathcal{A}} \times C_{\mathcal{A}}$. *Furthermore,* $\mathcal{A}$ *is self-companion* $\Leftrightarrow$ $\mathcal{F}$ *is self-companion* $\Leftrightarrow$ $\mathcal{G}$ *is self-companion.*

Proof. By definition, $P_{\mathcal{F}} = P_{\mathcal{A}}$, $P_{\mathcal{G}} = C_{\mathcal{A}}$, and $C_{\mathcal{F}} = P_{\mathcal{A}} \times C_{\mathcal{A}} = C_{\mathcal{G}}$. To show that $(\mathcal{F}, \mathcal{G})$ is a parametrization, we need to show that $d_{\mathcal{G}} \circ e_{\mathcal{F}} = e_{\mathcal{A}}$, $d_{\mathcal{F}} \circ e_{\mathcal{G}} = d_{\mathcal{A}}$, and that $\mathcal{F}$ and $\mathcal{G}$ are codes.

By examining Algorithm 8.2, we see that $\mathcal{F}$ is a code since the only possible edge incident on $(\pi, \kappa) \in C_{\mathcal{F}}$ in $e_{\mathcal{F}}$ or $d_{\mathcal{F}}^{nv}$ is $\{(\pi, (\pi, \kappa))\}$. We similarly see that $\mathcal{G}$ is a code. Since

$$(\pi, \kappa) \in e_{\mathcal{A}} \Leftrightarrow (\pi, (\pi, \kappa)) \in e_{\mathcal{F}} \text{ and } (\kappa, (\pi, \kappa)) \in d_{\mathcal{G}}^{nv}$$

$$\Leftrightarrow (\pi, \kappa)) \in d_{\mathcal{G}} \circ e_{\mathcal{F}}$$

and

$$(\kappa, \pi) \in d_{\mathcal{A}} \Leftrightarrow (\pi, (\pi, \kappa)) \in d_{\mathcal{F}}^{nv} \text{ and } (\kappa, (\pi, \kappa)) \in e_{\mathcal{G}}$$

$$\Leftrightarrow (\kappa, \pi) \in d_{\mathcal{F}} \circ e_{\mathcal{G}},$$

then $d_{\mathcal{G}} \circ e_{\mathcal{F}} = e_{\mathcal{A}}$ and $d_{\mathcal{F}} \circ e_{\mathcal{G}} = d_{\mathcal{A}}$.

Notice that $\mathcal{A}$ is self-companion $\Leftrightarrow$ both $\mathcal{F}$ and $\mathcal{G}$ are self-companion since

$$(\pi, \kappa) \in e_{\mathcal{A}} \cap d_{\mathcal{A}}^{nv} \Leftrightarrow (\pi, (\pi, \kappa)) \in e_{\mathcal{F}} \cap d_{\mathcal{F}}^{nv} \text{ and } (\kappa, (\pi, \kappa)) \in e_{\mathcal{G}} \cap d_{\mathcal{G}}^{nv}.$$

Furthermore,

$$(\pi, (\pi, \kappa)) \in e_{\mathcal{F}} \Leftrightarrow (\kappa, (\pi, \kappa)) \in d_{\mathcal{G}}^{nv} \text{ and } (\pi, (\pi, \kappa)) \in d_{\mathcal{F}}^{nv} \Leftrightarrow (\kappa, (\pi, \kappa)) \in e_{\mathcal{G}}.$$

Thus, $\mathcal{F}$ is self-companion if and only if $\mathcal{G}$ is self-companion. $\qquad\square$

The following example shows that parameterizations are not, in general, unique. It also shows that the parametrization given in Algorithm 8.2 is not minimal in any sense.

Example 8.4. *Let* $\mathcal{A} = (P_{\mathcal{A}} = \{1, 2, 3\}, C_{\mathcal{A}} = \{a\}, e_{\mathcal{A}} = \{(1, a), (2, a), (3, a)\}, d_{\mathcal{A}} = e_{\mathcal{A}}^{nv})$ *as depicted in Figure 13.*
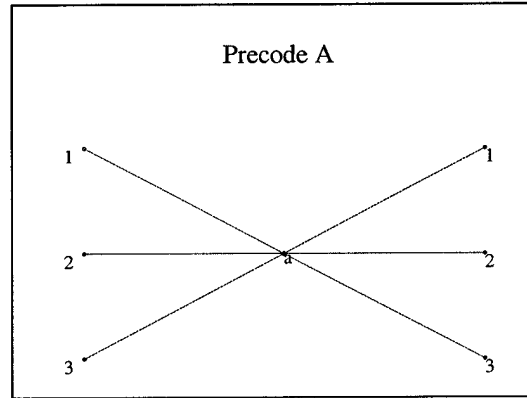
FIGURE 13. The Precode $\mathcal{A}$ in Example 8.4

Consider $\mathcal{F}' = (\{1, 2, 3\}, \{x, y\}, e_{\mathcal{F}'} = \{(1, x), (2, x), (3, x)\}, d_{\mathcal{F}'} = \{(y, 1), (y, 2), (y, 3)\})$ *and* $\mathcal{G}' = (\{a\}, \{x, y\}, e_{\mathcal{G}'} = \{(a, y))\}, d_{\mathcal{G}'} = \{(x, a)\})$ *as depicted in Figure 14.*
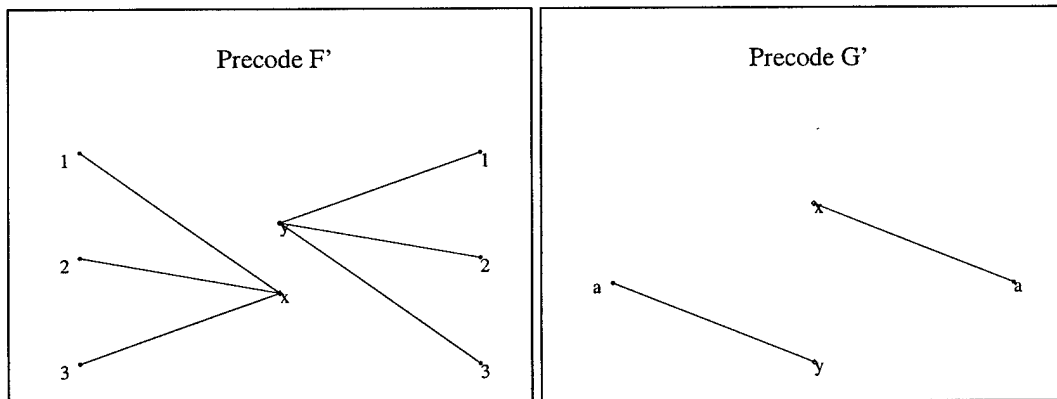
FIGURE 14. A Parametrization of $\mathcal{A}$

Then $(\mathcal{F}', \mathcal{G}')$ is a parametrization of $\mathcal{A}$ in $\{x, y\}$. However, the parametrization $(\mathcal{F}, \mathcal{G})$ of $\mathcal{A}$ given in Algorithm 8.2 satisfies

$$\mathcal{F} = (\{1, 2, 3\}, \{(1, a), (2, a), (3, a)\}, e_{\mathcal{F}} = \{(1, (1, a)), (2, (2, a)), (3, (3, a))\}, d_{\mathcal{F}} = e_{\mathcal{F}}^{nv})$$

and

$$\mathcal{G} = (\{a\}, \{(1, a), (2, a), (3, a)\}, e_{\mathcal{G}} = \{(a, (1, a)), (a, (2, a)), (a, (3, a))\}, d_{\mathcal{G}} = e_{\mathcal{G}}^{nv}).$$

Note that in Figure 15, we use $1a$ to represent $(1, a)$, etc.
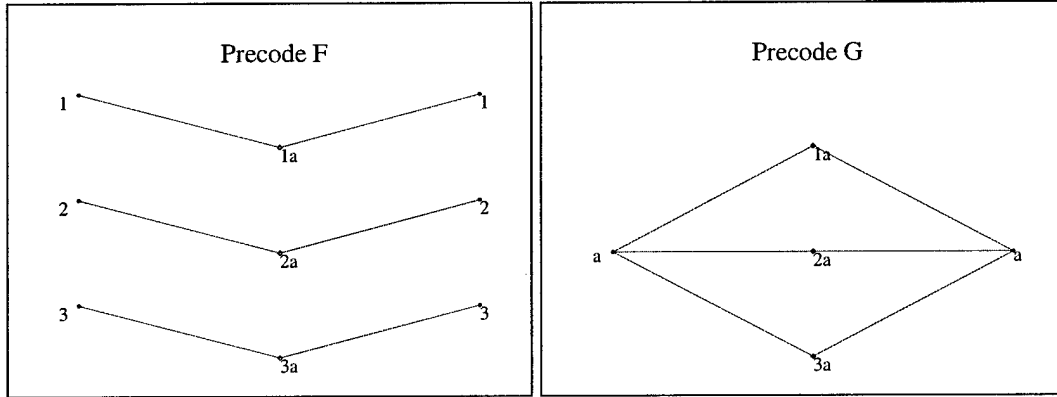
FIGURE 15. The Parametrization of $\mathcal{A}$ from Algorithm 8.2

Note that $\mathcal{F}$ and $\mathcal{G}$ have $|P_A \times C_A| = 3$ codetext elements, while $\mathcal{F}'$ and $\mathcal{G}'$ have 2. Furthermore, $|e_{\mathcal{F}}| = |d_{\mathcal{F}}| = |e_{\mathcal{G}}| = |d_{\mathcal{G}}| = |e_{\mathcal{F}'}| = |d_{\mathcal{F}'}| = 3$, while $|e_{\mathcal{G}'}| = |d_{\mathcal{G}'}| = 1$.

Although it is not minimal, the parametrization given in Algorithm 8.2 is nice in that it preserves the self-companion property.

8.2. **An ED-Split Parametrization.** We now attempt to find a parametrization which is minimal in some sense. It turns out that $\mathcal{F}'$ in Example 8.4 is isomorphic to $\mathcal{A}_{\text{led}}$ recall (Algorithm 7.1). This motivates the following algorithm.

Algorithm 8.5. *Let $\mathcal{A}$ be a precode. We construct a parametrization*

$$(F = (P_{\mathcal{F}} = P_{\mathcal{A}}, C_{\mathcal{F}}, e_{\mathcal{F}}, d_{\mathcal{F}}), G = (P_{\mathcal{G}} = C_{\mathcal{A}}, C_{\mathcal{G}}, e_{\mathcal{G}}, d_{\mathcal{G}}))$$

of $\mathcal{A}$ such that $\mathcal{F}$ is isomorphic to the subcode of $\mathcal{A}_{\mathsf{ledl}}$ which is formed by removing the isolated vertices from $C_{\mathcal{A}_{\mathsf{ledl}}}$. By relabling if necessary, we assume that $P_{\mathcal{A}}$ and $C_{\mathcal{A}}$ are disjoint. We further assume that no element in $P_{\mathcal{A}}$ is of the form a, $_{in}a$, or a_{out} for any $a \in C_{\mathcal{A}}$. We construct $\mathcal{F} = \mathcal{A}_{\mathsf{ledl}}$ as in Algorithm 7.1, with the exception that we exclude the isolated vertices in $C_{\mathcal{A}}$ from $C_{\mathcal{F}}$. We now give the algorithm for constructing $\mathcal{G}$:

Set $P_{\mathcal{G}} = C_{\mathcal{A}}$, $C_{\mathcal{G}} = C_{\mathcal{F}}$, $e_{\mathcal{G}} = \emptyset$, and $d_{\mathcal{G}} = \emptyset$. For each $\kappa \in C_{\mathcal{A}}$

   If $\hat{\kappa} \in C_{\mathcal{F}}$, then

      / κ was not a split vertex */*

      For each $(\pi, \hat{\kappa}) \in e_{\mathcal{F}}$

         Add $(\hat{\kappa}, \kappa)$ to $d_{\mathcal{G}}$

      end for

      For each $(\hat{\kappa}, \pi) \in d_{\mathcal{F}}$

         Add $(\kappa, \hat{\kappa})$ to $e_{\mathcal{G}}$

      end for

   else

      / κ was a split vertex, and thus $_{in}\kappa, \kappa_{out} \in C_{\mathcal{F}}$. */*

      For each $(\pi, _{in}\kappa) \in e_{\mathcal{F}}$

         Add $(_{in}\kappa, \kappa)$ to $d_{\mathcal{G}}$

      end for

      For each $(\kappa_{out}, \pi) \in d_{\mathcal{F}}$

         Add (κ, κ_{out}) to $e_{\mathcal{G}}$

      end for

   end if

Note that we have made a substantial improvement since $|T| = |P_{\mathcal{A}}| \cdot |C_{\mathcal{A}}|$ in Algorithm 8.2, while $|T| \leq 2 \cdot |C_{\mathcal{A}}|$ in Algorithm 8.5.

8.3. A Minimal ED Parametrization. We begin with a definition.

Definition 8.7. *Let $\mathcal{A}$ be a precode, and let*

$$(\mathcal{F} = (P_{\mathcal{F}} = P_{\mathcal{A}}, C_{\mathcal{F}} = T, d_{\mathcal{F}}, d_{\mathcal{F}}), \mathcal{G} = (P_{\mathcal{G}} = C_{\mathcal{A}}, C_{\mathcal{G}} = T, e_{\mathcal{G}}, d_{\mathcal{G}}))$$

be a parametrization of $\mathcal{A}$. If $(\pi, t) \in e_{\mathcal{F}}$ and $(t, \kappa) \in d_{\mathcal{G}}$, then t is said to induce $(\pi, \kappa) \in e_{\mathcal{A}}$. We also say that the edges $(\pi, t) \in e_{\mathcal{F}}$ and $(t, \kappa) \in d_{\mathcal{G}}$ induce $(\pi, \kappa) \in e_{\mathcal{A}}$. Similarly, if $(t, \pi) \in d_{\mathcal{F}}$ and $(\kappa, t) \in e_{\mathcal{G}}$, then t is said to induce $(\kappa, \pi) \in d_{\mathcal{A}}$. We also say that the edges $(t, \pi) \in d_{\mathcal{F}}$ and $(\kappa, t) \in e_{\mathcal{G}}$ induce $(\kappa, \pi) \in d_{\mathcal{A}}$.

Since we are interested in finding a minimal parametrization for $\mathcal{A}$, then we may assume that T contains no elements which are isolated in $\mathcal{F}$ and $\mathcal{G}$.

Lemma 8.8. *Suppose that $\kappa \in C_{\mathcal{A}}$ is a split vertex in Algorithm 7.1, and let*

$$(\mathcal{F} = (P_{\mathcal{F}} = P_{\mathcal{A}}, C_{\mathcal{F}} = T, d_{\mathcal{F}}, d_{\mathcal{F}}), \mathcal{G} = (P_{\mathcal{G}} = C_{\mathcal{A}}, C_{\mathcal{G}} = T, e_{\mathcal{G}}, d_{\mathcal{G}}))$$

be any parametrization of $\mathcal{A}$. Then at least two elements in T are necessary to induce the edges incident on κ in $\mathcal{A}$—one for the edges in $e_{\mathcal{A}}$ and one for the edges in $d_{\mathcal{A}}$.

Proof. By the definition of a split vertex, there exist $\pi_1, \pi_2 \in P_{\mathcal{A}}$ with $\pi_1 \neq \pi_2$ such that $(\pi_1, \kappa) \in e_{\mathcal{A}}$ and $(\kappa, \pi_2) \in d_{\mathcal{A}}$. By the definition of a parametrization, there must be some $x \in T$ such that $(\pi_1, x) \in e_{\mathcal{F}}$ and $(x, \kappa) \in d_{\mathcal{G}}$. Similarly, there must be some $y \in T$ such that $(y, \pi_2) \in d_{\mathcal{F}}$ and $(\kappa, y) \in e_{\mathcal{G}}$. Since $\mathcal{F}$ must be a code, then we must have that $x \neq y$. $\square$

The above lemma shows that the parametrization in Algorithm 8.5 is minimal in a local sense. However, it may be possible to produce another parametrization of $\mathcal{A}$ from the parametrization in Algorithm 8.5 by combining certain elements in T which come from different elements in $C_{\mathcal{A}}$. We now explore this idea.

Notation 8.9. *Let A be a precode. For each non-isolated $\kappa \in C_A$, let $In_\kappa = \{\pi \in P_A \mid (\pi, \kappa) \in e_A\}$ and $Out_\kappa = \{\pi \in P_A \mid (\kappa, \pi) \in d_A\}$.*

Lemma 8.10. *Let $(\mathcal{F} = (P_\mathcal{F} = P_A, C_\mathcal{F} = T, e_\mathcal{F}, d_\mathcal{F}), \mathcal{G} = (P_\mathcal{G} = C_A, C_\mathcal{G} = T, e_\mathcal{G}, d_\mathcal{G}))$ be a parametrization of A. Suppose $\kappa_1, \kappa_2 \in C_A$ are non-isolated vertices with $\kappa_1 \neq \kappa_2$. Then*

1) If $\pi_1 \in In_{\kappa_1} \backslash In_{\kappa_2}$ and $x \in T$ induces $(\pi_1, \kappa_1) \in e_A$, then $x \neq y$ for any $y \in T$ which induces any edge incident on κ_2 in A.

2) If $\pi_1 \in Out_{\kappa_1} \backslash Out_{\kappa_2}$ and $x \in T$ induces $(\kappa_1, \pi_1) \in d_A$, then $x \neq y$ for any $y \in T$ which induces any edge incident on κ_2 in A.

Proof. We prove 1. The proof for 2 is similar. Suppose that $\pi_1 \in In_{\kappa_1} \backslash In_{\kappa_2}$ and that $x \in T$ induces $(\pi_1, \kappa_1) \in e_A$. Thus, $(\pi_1, x) \in e_\mathcal{F}$ and $(x, \kappa_1) \in d_\mathcal{G}$. Since $\pi_1 \in In_{\kappa_1} \backslash In_{\kappa_2}$, then $(\pi_1, \kappa_2) \notin e_A$. Suppose that $y \in T$ induces some edge in A incident on κ_2.

Case 1: Suppose that y induces $(\pi_2, \kappa_2) \in e_A$; that is, suppose that $(\pi_2, y) \in e_\mathcal{F}$ and $(y, \kappa_2) \in d_\mathcal{G}$. If $x = y$, then $(\pi_1, x) \in e_\mathcal{F}$ and $(y, \kappa_2) \in d_\mathcal{G}$ imply that $(\pi_1, \kappa_2) \in d_\mathcal{G} \circ e_\mathcal{F} = e_A$, a contradiction.

Case 2: Suppose that y induces $(\kappa_2, \pi_2) \in d_A$; that is, suppose that $(y, \pi_2) \in d_\mathcal{F}$ and $(\kappa_2, y) \in e_\mathcal{G}$. Now, $(\kappa_2, y) \in e_\mathcal{G}$ and $(x, \kappa_1) \in d_\mathcal{G}$ imply that $x \neq y$ since $\kappa_1 \neq \kappa_2$ and $\mathcal{G}$ is a code. $\square$

Notation 8.11. *Let A be a precode. For each nonempty $B \subseteq P_A$, let $In(A, B) = \{\kappa \in C_A \mid In_\kappa = B\}$ and $Out(A, B) = \{\kappa \in C_A \mid Out_\kappa = B\}$. Notice $\{In(A, B) \mid \emptyset \neq B \subseteq P_A\}$ forms a partition of the set of non-isolated elements in C_A, as does $\{Out(A, B) \mid \emptyset \neq B \subseteq P_A\}$.*

Algorithm 8.12. *Let A be a precode. We construct a parametrization*

$$(\mathcal{F} = (P_\mathcal{F} = P_A, C_\mathcal{F}, e_\mathcal{F}, d_\mathcal{F}), \mathcal{G} = (P_\mathcal{G} = C_A, C_\mathcal{G}, e_\mathcal{G}, d_\mathcal{G}))$$

of A in $T = A_{In} \cup A_{Out}$, where

$$A_{In} = \{i_B \mid \emptyset \neq B \subseteq P_A \text{ and } In(A, B) \neq \emptyset\} \text{ and } A_{Out} = \{o_B \mid \emptyset \neq B \subseteq P_A \text{ and } Out(A, B) \neq \emptyset\}.$$

Set $P_{\mathcal{F}} = P_A$, $P_{\mathcal{G}} = C_A$, $C_{\mathcal{F}} = T$, $C_{\mathcal{G}} = T$, $e_{\mathcal{F}} = \emptyset$, $d_{\mathcal{F}} = \emptyset$, $e_{\mathcal{G}} = \emptyset$, and $d_{\mathcal{G}} = \emptyset$.

For each $(\pi, \kappa) \in e_A$

    Let $B = In_\kappa$.

    Add (π, i_B) to $e_{\mathcal{F}}$

    Add (i_B, κ) to $d_{\mathcal{G}}$

end for

For each $(\kappa, \pi) \in d_A$

    Let $B = Out_\kappa$.

    Add (o_B, π) to $d_{\mathcal{F}}$

    Add (κ, o_B) to $e_{\mathcal{G}}$

end for

Definition 8.13. *A parametrization $(\mathcal{F} = (P_{\mathcal{F}}, C_{\mathcal{F}} = T, e_{\mathcal{F}}, d_{\mathcal{F}}), \mathcal{G} = (P_{\mathcal{G}}, C_{\mathcal{G}} = T, e_{\mathcal{G}}, d_{\mathcal{G}}))$ of a precode A in which both $\mathcal{F}$ and $\mathcal{G}$ are codes of a certain pre-defined type, α, is called an α-parametrization of A in T. For example, if both $\mathcal{F}$ and $\mathcal{G}$ are ED codes as defined in Definition 4.8, we call $(\mathcal{F}, \mathcal{G})$ an ED-parametrization of A.*

An α-parametrization of A in T is said to be a minimally-edged α-parametrization of A if each edge in A is induced by precisely one element in T and each edge in $\mathcal{F}$ and each edge in $\mathcal{G}$ induces some edge in A.

A parametrization $(\mathcal{F}, \mathcal{G})$ of A in T is said to be a minimal parametrization of A if it is minimally-edged and for any parametrization $(\mathcal{F}', \mathcal{G}')$ of A in S, we have $|T| \leq |S|$, $|e_{\mathcal{F}}| \leq |e_{\mathcal{F}'}|$, $|d_{\mathcal{F}}| \leq |d_{\mathcal{F}'}|$, $|e_{\mathcal{G}}| \leq |e_{\mathcal{G}'}|$, and $|d_{\mathcal{G}}| \leq |d_{\mathcal{G}'}|$.

Theorem 8.14. *Let A be a precode, and let $\mathcal{F}$ and $\mathcal{G}$ be as defined in Algorithm 8.12. Then $(\mathcal{F}, \mathcal{G})$ is an ED-parametrization of A in $A_{In} \cup A_{Out}$. Furthermore, it is a minimal ED-parametrization of A.*

Proof. By definition, $P_{\mathcal{F}} = P_{\mathcal{A}}$, $P_{\mathcal{G}} = C_{\mathcal{A}}$, and $C_{\mathcal{F}} = A_{In} \cup A_{Out} = C_{\mathcal{G}}$. To show that $(\mathcal{F}, \mathcal{G})$ is a parametrization, we need only show that $d_{\mathcal{G}} \circ e_{\mathcal{F}} = e_{\mathcal{A}}$, $d_{\mathcal{F}} \circ e_{\mathcal{G}} = d_{\mathcal{A}}$, and that $\mathcal{F}$ and $\mathcal{G}$ are codes.

By examining Algorithm 8.12, we see that the only edges incident in $\mathcal{F}$ on $i_B \in C_{\mathcal{F}}$ are encode edges, and the only edges incident in $\mathcal{F}$ on $o_B \in C_{\mathcal{F}}$ are decode edges. Thus, $\mathcal{F}$ is an ED-code. We similarly see that $\mathcal{G}$ is an ED-code.

Let $\kappa \in C_{\mathcal{A}}$, $B_1 = In_\kappa$, and $B_2 = Out_\kappa$. Then, $d_{\mathcal{G}} \circ e_{\mathcal{F}} = e_{\mathcal{A}}$ and $d_{\mathcal{F}} \circ e_{\mathcal{G}} = d_{\mathcal{A}}$ since

$$(\pi, \kappa) \in e_{\mathcal{A}} \Leftrightarrow (\pi, i_{B_1}) \in e_{\mathcal{F}} \text{ and } (i_{B_1}, \kappa) \in d_{\mathcal{G}}$$

$$\Leftrightarrow (\pi, \kappa) \in d_{\mathcal{G}} \circ e_{\mathcal{F}}.$$

and

$$(\kappa, \pi) \in d_{\mathcal{A}} \Leftrightarrow (o_{B_2}, \pi) \in d_{\mathcal{F}} \text{ and } (\kappa, o_{B_2}) \in e_{\mathcal{G}}$$

$$\Leftrightarrow (\kappa, \pi) \in d_{\mathcal{F}} \circ e_{\mathcal{G}}.$$

We now show minimality. By Algorithm 8.12, each edge in $\mathcal{A}$ is induced by precisely one element in T. We now show that for any vertex in T, there must be at least one corresponding vertex in S for any ED-parametrization

$$(\mathcal{F}' = (P_{\mathcal{F}'}, C_{\mathcal{F}'} = S, e_{\mathcal{F}'}, d_{\mathcal{F}'}), \mathcal{G}' = (P_{\mathcal{G}'}, C_{\mathcal{G}'} = S, e_{\mathcal{G}'}, d_{\mathcal{G}'}))$$

of $\mathcal{A}$. Recall that T contains no isolated vertices.

Let $K_1 \subseteq C_{\mathcal{A}}$ be the set of all codetext elements which are part of an edge in $e_{\mathcal{A}}$ and $K_2 \subseteq C_{\mathcal{A}}$ be the set of all codetext elements which are part of an edge in $d_{\mathcal{A}}$. Since $(\mathcal{F}', \mathcal{G}')$ must be an ED-parametrization, then each element in S can either induce only edges in $e_{\mathcal{A}}$ or only edges in $d_{\mathcal{A}}$. Let $S_1 \subseteq S$ be the set of elements which induce edges in $e_{\mathcal{A}}$ and $S_2 \subseteq S$ be the set of elements which induce edges in $d_{\mathcal{A}}$. Note that S_1 and S_2 are disjoint.

For each $\kappa \in K_1$, there must be some $\hat{\kappa} \in S_1$ which induces an edge in $e_{\mathcal{A}}$ incident on κ. Furthermore, by Lemma 8.10 (part 1), we may only have $\hat{\kappa} = \hat{\gamma}$ for some $\kappa, \gamma \in C_{\mathcal{A}}$ only if $In_\kappa = In_\gamma$. Thus, there must be at least one element in S_1 for each element in $\mathcal{A}_{In}$. Similarly,

by applying Lemma 8.10 (part 2), we see that there must be at least one element in S_2 for each element in A_{Out}. Since $T = A_{In} \cup A_{Out}$, then $|T| \leq |S|$. $\qquad\square$

We now give examples showing that sometimes the parametrization in Algorithm 8.5 is "smaller" than the one given in Algorithm 8.12, while sometimes the opposite holds.

Example 8.15. *Let* $\mathcal{A} = (P_{\mathcal{A}} = \{1\}, C_{\mathcal{A}} = \{a\}, e_{\mathcal{A}} = \{(1,a)\}, d_{\mathcal{A}} = e_{\mathcal{A}}^{nv})$, *as depicted in Figure 16.*

FIGURE 16. The Precode $\mathcal{A}$ from Example 8.15

The parametrization $(\mathcal{F}, \mathcal{G})$ *of* $\mathcal{A}$ *given in Algorithm 8.5 satisfies* $P_{\mathcal{F}} = P_{\mathcal{A}}, P_{\mathcal{G}} = C_{\mathcal{A}}$, $C_{\mathcal{F}} = \{a\} = C_{\mathcal{G}}$, $e_{\mathcal{F}} = \{(1,a)\}$, $d_{\mathcal{F}} = e_{\mathcal{F}}^{nv}$, $e_{\mathcal{G}} = \{(a,a)\}$, *and* $d_{\mathcal{G}} = e_{\mathcal{G}}^{nv}$ *as depicted in Figure 17.*

FIGURE 17. The Parametrization Given by Algorithm 8.5 (Example 8.15)

The parametrization $(\mathcal{F}', \mathcal{G}')$ of $\mathcal{A}$ given in Algorithm 8.12 satisfies $P_{\mathcal{F}'} = P_{\mathcal{A}}$, $P_{\mathcal{G}'} = C_{\mathcal{A}}$, $C_{\mathcal{F}'} = \{i_{\{1\}}, o_{\{1\}}\} = C_{\mathcal{G}'} = \{i_{\{1\}}, o_{\{1\}}\}$, $e_{\mathcal{F}'} = \{(1, i_{\{1\}})\}$, $d_{\mathcal{F}'} = \{(o_{\{1\}}, 1)\})$, $e_{\mathcal{G}'} = \{(a, o_{\{1\}})\}$, and $d_{\mathcal{G}'} = \{(i_{\{1\}}, a)\})$. In Figure 18, we use $o\_\{1\}$ to represent $o_{\{1\}}$, etc.
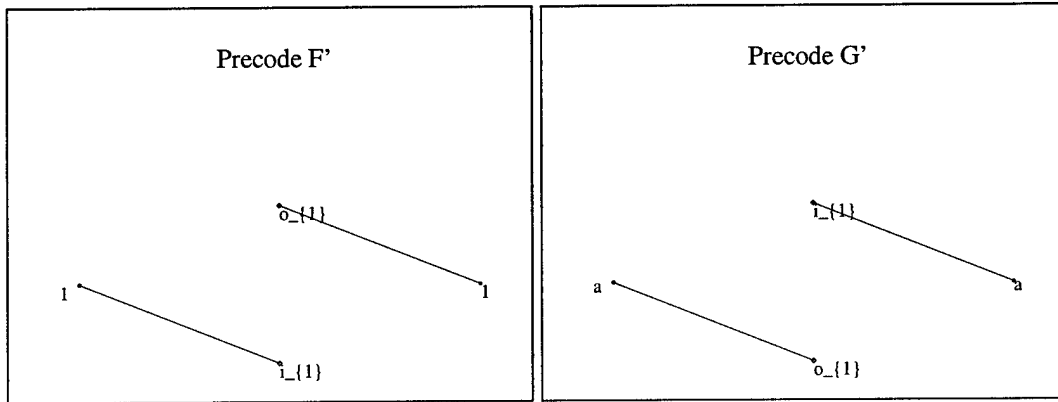
FIGURE 18. The Parametrization Given by Algorithm 8.12 (Example 8.15)

Note that $\mathcal{F}$ and $\mathcal{G}$ have 1 codetext element, while $\mathcal{F}'$ and $\mathcal{G}'$ have 2.

Example 8.16. Let $\mathcal{A} = (P_{\mathcal{A}} = \{1\}, C_{\mathcal{A}} = \{a, b\}, e_{\mathcal{A}} = \{(1, a), (1, b)\}, d_{\mathcal{A}} = \{(a, 1)\})$. See Figure 19.

FIGURE 19. The Precode $\mathcal{A}$ from Example 8.16

The parametrization $(\mathcal{F}, \mathcal{G})$ of $\mathcal{A}$ given in Algorithm 8.5 satisfies $P_{\mathcal{F}} = P_{\mathcal{A}}$, $P_{\mathcal{G}} = C_{\mathcal{A}}$, $C_{\mathcal{F}} = \{a,b\} = C_{\mathcal{G}}$, $e_{\mathcal{F}} = \{(1,a),(1,b)\}$, $d_{\mathcal{F}} = \{(a,1)\}$, $e_{\mathcal{G}} = \{(a,a)\}$, and $d_{\mathcal{G}} = \{(a,a),(b,b)\}$. See Figure 20.
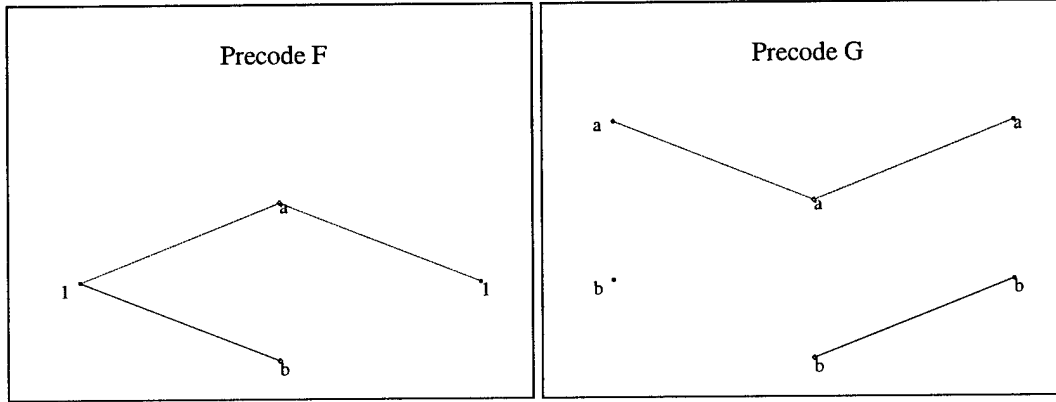
FIGURE 20. The Parametrization Given by Algorithm 8.5 (Example 8.16)

The parametrization $(\mathcal{F}', \mathcal{G}')$ of $\mathcal{A}$ given in Algorithm 8.12 satisfies $P_{\mathcal{F}'} = P_{\mathcal{A}}$, $P_{\mathcal{G}'} = C_{\mathcal{A}}$, $C_{\mathcal{F}'} = \{i_{\{1\}}, o_{\{1\}}\} = C_{\mathcal{G}'}$, $e_{\mathcal{F}'} = \{(1, i_{\{1\}})\}$, $d_{\mathcal{F}'} = \{(o_{\{1\}}, 1)\}$, $e_{\mathcal{G}'} = \{(a, o_{\{1\}})\}$, and $d_{\mathcal{G}'} = \{(i_{\{1\}}, a), (i_{\{1\}}, b)\}$. See Figure 21.

FIGURE 21. The Parametrization Given by Algorithm 8.12 (Example 8.16)

Note that $|C_{\mathcal{F}}| = |C_{\mathcal{F}'}|$. Also, $|d_{\mathcal{F}}| = |d_{\mathcal{F}'}| = 1$, $|e_{\mathcal{G}}| = |e_{\mathcal{G}'}| = 1$, and $|d_{\mathcal{G}}| = |d_{\mathcal{G}'}| = 2$. However, $|e_{\mathcal{F}}| = 2$, while $|e_{\mathcal{F}'}| = 1$.

Example 8.17. Let $\mathcal{A} = (P_A = \{1\}, C_A = \{a,b,c\}, e_A = \{(1,a),(1,b),(1,c)\}, d_A = e_A^{nv})$, as depicted in Figure 22.
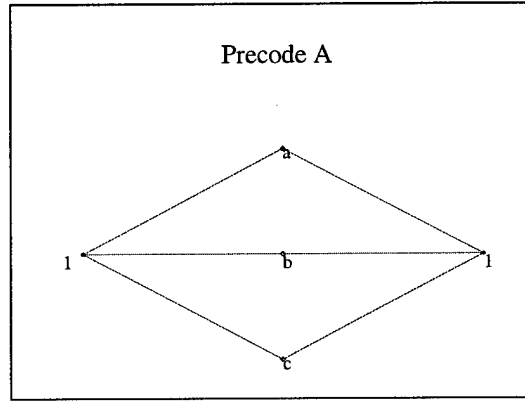
FIGURE 22. The Precode $\mathcal{A}$ from Example 8.17

The parametrization $(\mathcal{F}, \mathcal{G})$ of $\mathcal{A}$ given in Algorithm 8.5 satisfies $P_{\mathcal{F}} = P_A$, $P_{\mathcal{G}} = C_A$, $C_{\mathcal{F}} = \{a,b,c\} = C_{\mathcal{G}}$, $e_{\mathcal{F}} = \{(1,a),(1,b),(1,c)\}$, $d_{\mathcal{F}} = e_{\mathcal{F}}^{nv}$, $e_{\mathcal{G}} = \{(a,a),(b,b),(c,c)\}$, and $d_{\mathcal{G}} = e_{\mathcal{G}}^{nv}$. See Figure 23.
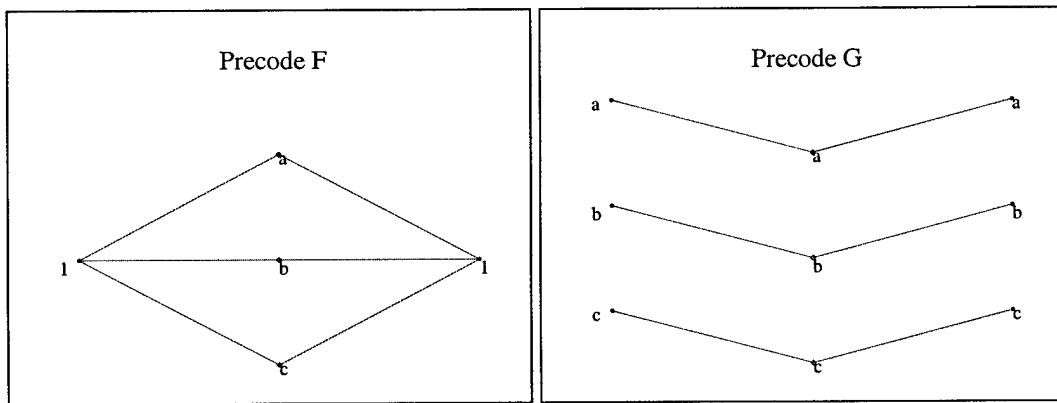
FIGURE 23. The Parametrization Given by Algorithm 8.5 (Example 8.17)

The parametrization $(F' = (P_{F'}, C_{F'}, e_{F'}, d_{F'}), G' = (P_{G'}, C_{G'}, e_{G'}, d_{G'}))$ of $\mathcal{A}$ given in Algorithm 8.12 satisfies $P_{\mathcal{F}'} = P_{\mathcal{A}}$, $P_{\mathcal{G}'} = C_{\mathcal{A}}$, $C_{\mathcal{F}'} = \{i_{\{1\}}, o_{\{1\}}\} = C_{\mathcal{G}'}$, $e_{\mathcal{F}'} = \{(1, i_{\{1\}})\}$, $d_{\mathcal{F}'} = \{(o_{\{1\}}, 1)\}$, $e_{\mathcal{G}'} = \{(a, o_{\{1\}}), (b, o_{\{1\}}), (c, o_{\{1\}})\}$, and $d_{\mathcal{G}'} = \{(i_{\{1\}}, a), (i_{\{1\}}, b), (i_{\{1\}}, c)\}$. See Figure 24.
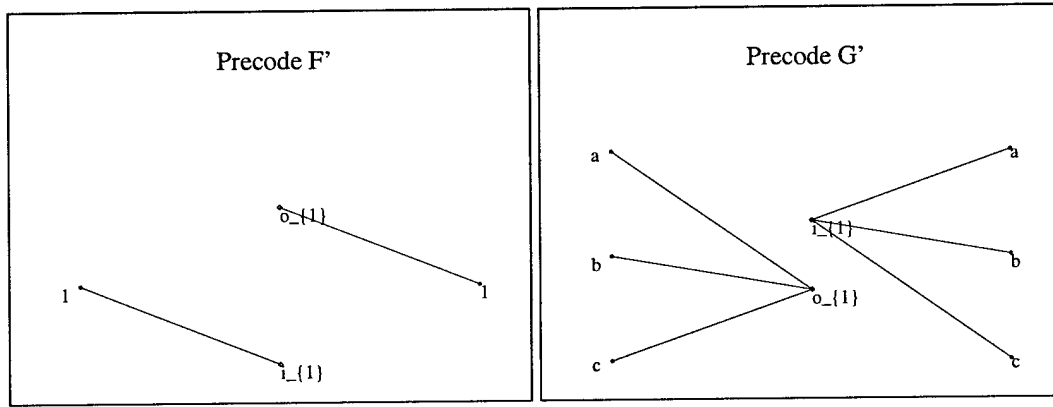
FIGURE 24. The Parametrization Given by Algorithm 8.12 (Example 8.17)

Note that $|e_{\mathcal{G}}| = |e_{\mathcal{G}'}| = 3$ and $|d_{\mathcal{G}}| = |d_{\mathcal{G}'}| = 3$. However, $|C_{\mathcal{F}}| = 3$, while $|C_{\mathcal{F}'}| = 2$. Also, $|e_{\mathcal{F}}| = |d_{\mathcal{F}}| = 3$, while $|e_{\mathcal{F}'}| = |d_{\mathcal{F}'}| = 1$.

Example 8.18. Let $\mathcal{A} = (P_{\mathcal{A}} = \{1, 2\}, C_{\mathcal{A}} = \{a, b, c\}, e_{\mathcal{A}} = \{(1, a), (1, b), (2, c)\}, d_{\mathcal{A}} = e_{\mathcal{A}}^{nv})$ as depicted in Figure 25.
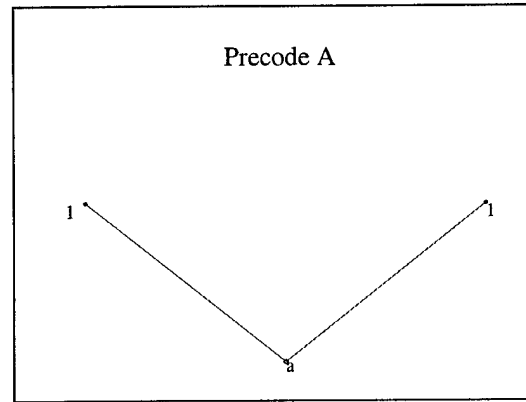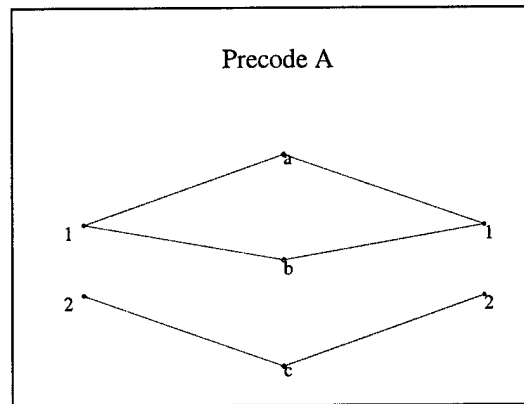
FIGURE 25. The Precode $\mathcal{A}$ from Example 8.18

The parametrization $(\mathcal{F}, \mathcal{G})$ of $\mathcal{A}$ given in Algorithm 8.5 satisfies $e_{\mathcal{F}} = \{(1,a),(1,b),(2,c)\}$, $d_{\mathcal{F}} = e_{\mathcal{F}}^{nv}$), $e_{\mathcal{G}} = \{(a,a),(b,b),(c,c)\}$, and $d_{\mathcal{G}} = e_{\mathcal{G}}^{nv}$. See Figure 26.
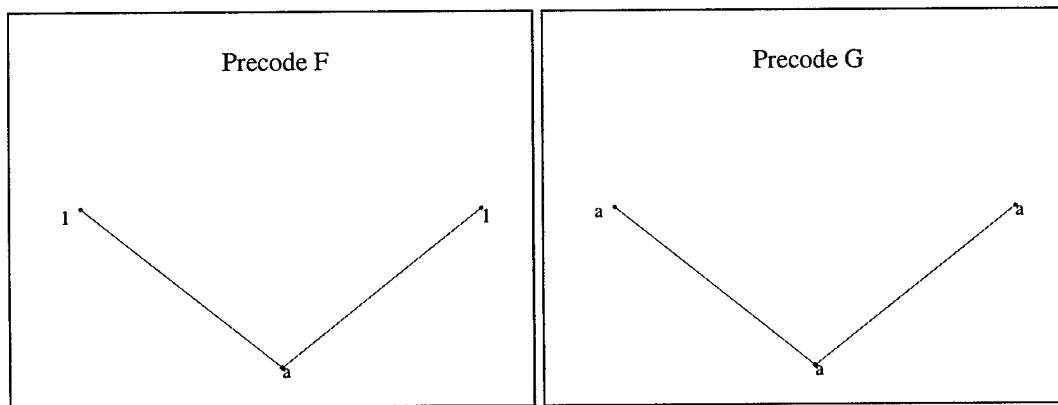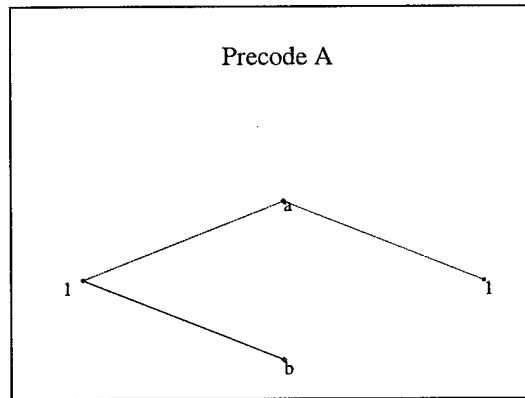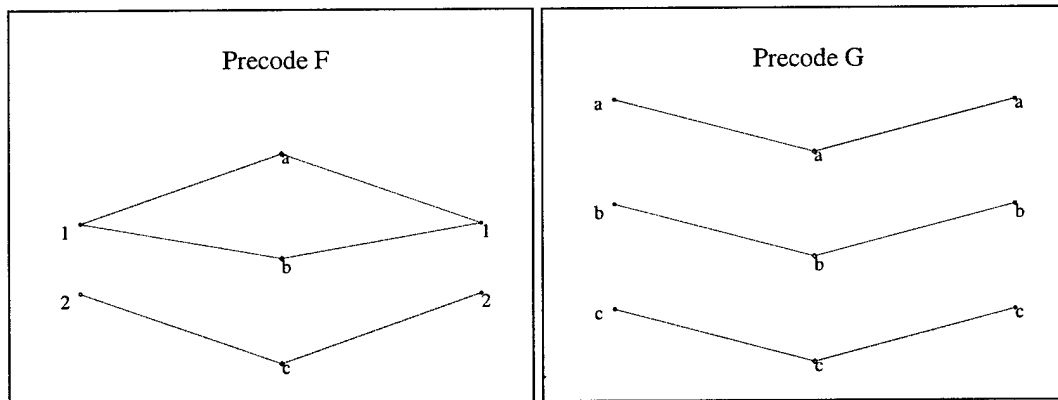

FIGURE 26. The Parametrization Given by Algorithm 8.5 (Example 8.18)

The parametrization $(\mathcal{F}', \mathcal{G}')$ of $\mathcal{A}$ given in Algorithm 8.12 satisfies $e_{\mathcal{F}'} = \{(1, i_{\{1\}}),(2, i_{\{2\}})\}$, $d_{\mathcal{F}'} = \{(o_{\{1\}}, 1),(o_{\{2\}}, 2)\})$, $e_{\mathcal{G}'} = \{(a, o_{\{1\}}),(b, o_{\{1\}}),(c, o_{\{2\}})\}$, and $d_{\mathcal{G}'} = \{(i_{\{1\}}, a),(i_{\{1\}}, b)\}),(i_{\{2\}}, c)\}$. See Figure 27.
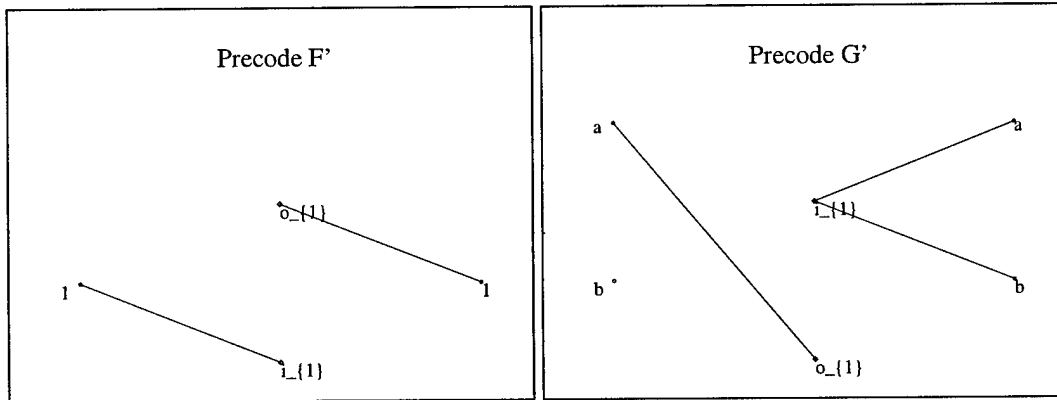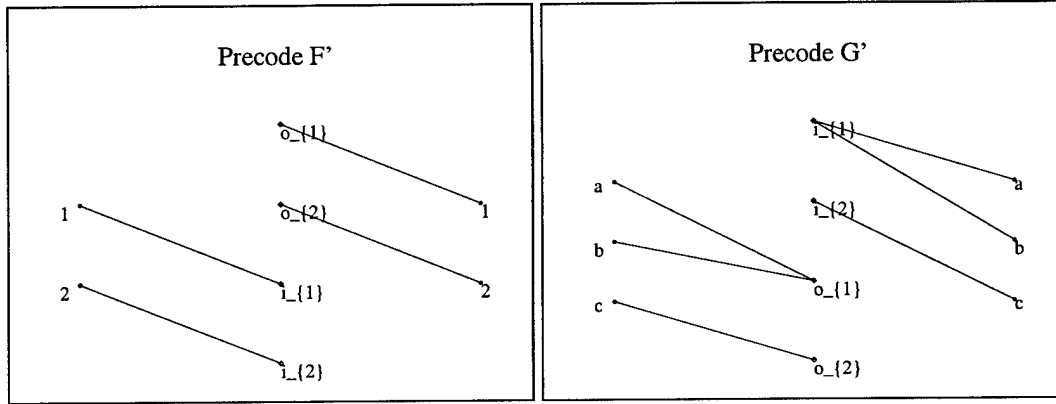
FIGURE 27. The Parametrization Given by Algorithm 8.12 (Example 8.18)

Note that $|e_{\mathcal{G}}| = |e_{\mathcal{G}'}| = 3$ and $|d_{\mathcal{G}}| = |d_{\mathcal{G}'}| = 3$. However, $|C_{\mathcal{F}}| = 3$, while $|C_{\mathcal{F}'}| = 4$. Also, $|e_{\mathcal{F}}| = |d_{\mathcal{F}}| = 3$, while $|e_{\mathcal{F}'}| = |d_{\mathcal{F}'}| = 2$. Thus, $(\mathcal{F}, \mathcal{G})$ is "smaller" in terms of number of codetext elements, while $(\mathcal{F}', \mathcal{G}')$ is "smaller" with respect to number of edges.

8.4. A Minimal Parametrization. Since we now better understand the properties of parameterizations, we continue to work toward creating a truly minimal one. Any parametrization of $\mathcal{A}$ is of the form

$$(\mathcal{F} = (P_{\mathcal{F}}, C_{\mathcal{F}} = T, e_{\mathcal{F}}, d_{\mathcal{F}}), \mathcal{G} = (P_{\mathcal{G}}, C_{\mathcal{G}} = T, e_{\mathcal{G}}, d_{\mathcal{G}})).$$

We attempt to construct a parametrization which minimizes $|T|$. We begin with a lemma that seems unmotivated, but is necessary for the discussion which follows.

Lemma 8.19. *Let $(\mathcal{F} = (P_{\mathcal{F}}, C_{\mathcal{F}} = T, e_{\mathcal{F}}, d_{\mathcal{F}}), \mathcal{G} = (P_{\mathcal{G}}, C_{\mathcal{G}} = T, e_{\mathcal{G}}, d_{\mathcal{G}}))$ be a parametrization of a precode $\mathcal{A}$ in T. Let $\kappa_1, \kappa_2 \in C_{\mathcal{A}}$ be non-isolated vertices such that $\kappa_1 \neq \kappa_2$, with κ_1 a codetext element of type s. Suppose that $t_1 \in T$ induces the edges incident on κ_1 and $t_2 \in T$ induces any edge incident on κ_2. Then $t_1 \neq t_2$.*

Proof. Since κ_1 is of type s, then the only edges incident on κ_1 are $(\pi_1, \kappa_1) \in e_{\mathcal{A}}$ and $(\kappa_1, \pi_1) \in d_{\mathcal{A}}$ for some $\pi_1 \in P_{\mathcal{A}}$. An edge incident on κ_2 may be of the form $(\pi_2, \kappa_2) \in e_{\mathcal{A}}$ or $(\kappa_2, \pi_2) \in d_{\mathcal{A}}$. W.l.o.g., we suppose that t_2 induces the edge $(\pi_2, \kappa_2) \in e_{\mathcal{A}}$.

Assume that $t_1 = t_2$. By Lemma 8.10, then $In_{\kappa_1} = In_{\kappa_2}$. Since $In_{\kappa_1} = \{\pi_1\}$, then $\pi_2 = \pi_1$. Since $t_2 = t_1$ induces $(\pi_2 = \pi_1, \kappa_2) \in e_{\mathcal{A}}$, then $(\pi_1, t_1) \in e_{\mathcal{F}}$ and $(t_1, \kappa_2) \in d_{\mathcal{G}}$. Also, $(\kappa_1, t_1) \in e_{\mathcal{G}}$ and $(t_1, \pi_1) \in d_{\mathcal{F}}$ since t_1 induces $(\kappa_1, \pi_1) \in d_{\mathcal{A}}$. Since $\mathcal{G}$ is a code, $(\kappa_1, t_1) \in e_{\mathcal{G}}$, and $(t_1, \kappa_2) \in d_{\mathcal{G}}$, then $\kappa_1 = \kappa_2$, a contradiction. $\qquad \square$

Let C_1 be the set of elements in $C_{\mathcal{A}}$ which must be split to form $\mathcal{A}_{\text{ledl}}$ (recall Algorithm 7.1). These are precisely the $\kappa \in C_{\mathcal{A}}$ for which there exist $\pi_1, \pi_2 \in P_{\mathcal{A}}$ with $\pi_1 \neq \pi_2$ such that $(\pi_1, \kappa) \in e_{\mathcal{A}}$ and $(\kappa, \pi_2) \in d_{\mathcal{A}}$. Let C_2 be the set of elements in $C_{\mathcal{A}}$ of type d or e, and let C_3 be the set of elements in $C_{\mathcal{A}}$ of type s. Then C_1, C_2, and C_3 form a partition of the non-isolated elements in $C_{\mathcal{A}}$.

If $\kappa \in C_1$, then by Lemma 8.8, at least two elements in T are necessary to induce the edges incident on κ in $\mathcal{A}$ (one for the encode edges and one for the decode edges). If $\kappa \in C_2$, it is possible to induce the edges incident on κ using only one element in T. The parameterizations in Algorithm 8.5 and Algorithm 8.12 are such parameterizations.

If $\kappa \in C_3$, then the only edges incident on κ are $(\pi, \kappa) \in e_{\mathcal{A}}$ and $(\kappa, \pi) \in d_{\mathcal{A}}$ for some $\pi \in P_{\mathcal{A}}$. In this case, it is possible to induce these edges with a single element in T (as in Algorithm 8.5). The other alternative (as employed in Algorithm 8.12) is to use two elements in T to induce these edges. Furthermore, Lemma 8.19 shows that if we use a single element in T to induce the edges on κ, then it cannot be used to induce edges on any other element in $C_{\mathcal{A}}$.

We have seen (as in Algorithm 8.12) that if we use distinct elements to induce the encode and decode edges incident on an arbitrary non-isolated $\kappa \in C_{\mathcal{A}}$, then it may be possible to use these same elements to induce the edges incident on other elements in $C_{\mathcal{A}}$. In particular, for any $\chi \in C_{\mathcal{A}}$ such that $In_{\kappa} = In_{\chi}$, we may use one element $t_1 \in T$ to induce all of the edges in $e_{\mathcal{A}}$ incident on κ and χ. Similarly, if $Out_{\kappa} = Out_{\chi}$, we may use one element $t_2 \in T$ to induce all of the edges in $d_{\mathcal{A}}$ incident on κ and χ. Furthermore, Lemma 8.10 shows that if $In_{\kappa} \neq In_{\chi}$, then t_1 cannot be used to induce both an edge in $e_{\mathcal{A}}$ incident on κ and an edge in $e_{\mathcal{A}}$ incident on χ. Similarly, if $Out_{\kappa} \neq Out_{\chi}$, then t_2 cannot be used to induce both an edge in $d_{\mathcal{A}}$ incident on κ and an edge in $d_{\mathcal{A}}$ incident on χ.

Now, if $\kappa \in C_3$, then using separate elements to induce the encode and decode edges incident on κ will result in fewer elements in T when we can use both of the elements to induce edges on other elements in C_A. If we can use only one of the elements to induce edges on other elements in C_A, then the number of elements required in T will remain the same, but the number of edges in $\mathcal{F}$ required to induce the edges incident on κ will be reduced. This is the gist of the following lemma.

Lemma 8.20. *Let A be a precode. Let $\kappa \in C_A$ be a non-isolated vertex of type s; that is, the edges incident on κ are $(\pi, \kappa) \in e_A$ and $(\kappa, \pi) \in d_A$ for some $\pi \in P_A$. Suppose there is some $\chi \neq \kappa$ for which $In_\kappa = In_\chi$ (resp. $Out_\kappa = Out_\chi$), and suppose that there is no $\chi' \in C_A$ for which $Out_\kappa = Out_{\chi'}$ (resp. $In_\kappa = In_{\chi'}$). Let*

$$\mathcal{B} = (P_A, C_\mathcal{B} = C_A \backslash \{\kappa\}, e_A \cap (P_A \times C_\mathcal{B}), d_A \cap (C_\mathcal{B} \times P_A));$$

that is, $\mathcal{B}$ is the precode formed from A be removing the codetext element κ. Let

$$(\mathcal{F} = (P_\mathcal{F}, C_\mathcal{F} = T, e_\mathcal{F}, d_\mathcal{F}), \mathcal{G} = (P_\mathcal{G}, C_\mathcal{G} = T, e_\mathcal{G}, d_\mathcal{G}))$$

be any parametrization of $\mathcal{B}$ in T such that $\chi_e \in T$ induces the elements in e_A incident on χ (resp. $\chi_d \in T$ induces the elements in d_A incident on χ). To create a parametrization for A from the parametrization $(\mathcal{F}, \mathcal{G})$ of $\mathcal{B}$, we must add an element, say $t \notin T$, to T to induce $(\kappa, \pi) \in d_A$ (resp. $(\pi, \kappa) \in e_A$).

Using t to generate only the edge $(\kappa, \pi) \in d_A$ (resp. $(\pi, \kappa) \in e_A$) will require fewer edges in $\mathcal{F}$ and $\mathcal{G}$ than will using t to induce both $(\pi, \kappa) \in e_A$ and $(\kappa, \pi) \in d_A$.

Proof. Suppose that we use t to induce both edges. We must add (π, t) to $e_\mathcal{F}$ and (t, κ) to $d_\mathcal{G}$ to generate $(\pi, \kappa) \in e_A$. We must add (κ, t) to $e_\mathcal{G}$ and (t, π) to $d_\mathcal{F}$ to generate $(\kappa, \pi) \in d_A$.

Now, suppose that we use χ_e to induce $(\pi, \kappa) \in e_A$ and t to induce $(\kappa, \pi) \in d_A$. Since χ_e generates the edges in e_A incident on χ and since $In_\chi = In_\kappa = \{\pi\}$, then $(\pi, \chi_e) \in e_\mathcal{F}$ and $(\chi_e, \kappa) \in d_\mathcal{G}$. To generate $(\kappa, \pi) \in d_A$, we need only add (κ, t) to $e_\mathcal{G}$ and (t, π) to $d_\mathcal{F}$. $\qquad\square$

Notation 8.21. *Let $\mathcal{A}$ be a precode. Recall the definitions made in Notation 8.11. Let*

$$S = \{\kappa \in C_{\mathcal{A}} \mid \kappa \text{ is of type } s\} \text{ and } L = \{\kappa \in S \mid |In(\mathcal{A}, In_\kappa)| = 1 \text{ and } |Out(\mathcal{A}, Out_\kappa)| = 1\}.$$

Recall that In_κ is the set of elements in $P_{\mathcal{A}}$ which are incident on κ via edges in $e_{\mathcal{A}}$. Also, recall that for each nonempty $B \subseteq P_{\mathcal{A}}$, $In(\mathcal{A}, B) = \{\kappa \in C_{\mathcal{A}} \mid In_\kappa = B\}$ and $Out(\mathcal{A}, B) = \{\kappa \in C_{\mathcal{A}} \mid Out_\kappa = B\}$. Thus, $In(\mathcal{A}, In_\kappa) = \{\chi \in C_{\mathcal{A}} \mid In_\kappa = In_\chi\}$ and $Out(\mathcal{A}, Out_\kappa) = \{\chi \in C_{\mathcal{A}} \mid Out_\kappa = Out_\chi\}$. Thus, L is the set of $\kappa \in C_{\mathcal{A}}$ for which using two elements to induce the edges incident on κ will not result in a "smaller" parametrization. Note that for $\emptyset \neq B \subseteq P_{\mathcal{A}}$, then

$$In(\mathcal{A}, B) \subseteq L \Leftrightarrow In(\mathcal{A}, B) = Out(\mathcal{A}, B) = \{\kappa\} \text{ for some } \kappa \in L$$

$$\Leftrightarrow Out(\mathcal{A}, B) \subseteq L.$$

Algorithm 8.22. *Let $\mathcal{A}$ be a precode. Let*

$$\mathcal{A}_{In} = \{i_B \mid \emptyset \neq B \subseteq P_{\mathcal{A}}, In(\mathcal{A}, B) \neq \emptyset, \text{ and } In(\mathcal{A}, B) \not\subseteq L\}$$

and

$$\mathcal{A}_{Out} = \{o_B \mid \emptyset \neq B \subseteq P_{\mathcal{A}}, Out(\mathcal{A}, B) \neq \emptyset, \text{ and } Out(\mathcal{A}, B) \not\subseteq L\}.$$

We construct a parametrization

$$(\mathcal{F} = (P_{\mathcal{F}} = P_{\mathcal{A}}, C_{\mathcal{F}} = T, e_{\mathcal{F}}, d_{\mathcal{F}}), \mathcal{G} = (P_{\mathcal{G}} = C_{\mathcal{A}}, C_{\mathcal{G}} = T, e_{\mathcal{G}}, d_{\mathcal{G}}))$$

of $\mathcal{A}$ in T, where $T = \mathcal{A}_{In_B} \cup \mathcal{A}_{Out_B} \cup L$. Note that $\mathcal{A}_{In_B}$, $\mathcal{A}_{Out_B}$, and L are pairwise disjoint. Except for the method of handling the edges incident on elements of L, this parametrization is identical to the one given in Algorithm 8.12.

Set $P_{\mathcal{F}} = P_{\mathcal{A}}$, $P_{\mathcal{G}} = C_{\mathcal{A}}$, $C_{\mathcal{F}} = T$, $C_{\mathcal{G}} = T$, $e_{\mathcal{F}} = \emptyset$, $d_{\mathcal{F}} = \emptyset$, $e_{\mathcal{G}} = \emptyset$, and $d_{\mathcal{G}} = \emptyset$.

For each non-isolated $\kappa \in C_{\mathcal{A}}$

  If $\kappa \in L$ then

   / κ is of type s */*

   Let $\pi \in P_{\mathcal{A}}$ be the unique element such that $(\pi, \kappa) \in e_{\mathcal{A}}$ and $(\kappa, \pi) \in d_{\mathcal{A}}$.

$Add\ (\pi, \kappa)\ to\ e_{\mathcal{F}}$

$Add\ (\kappa, \kappa)\ to\ d_{\mathcal{G}}$

$Add\ (\kappa, \pi)\ to\ d_{\mathcal{F}}$

$Add\ (\kappa, \kappa)\ to\ e_{\mathcal{G}}$

$else$

$Let\ B = In_{\kappa}.$

$For\ each\ (\pi, \kappa) \in e_{\mathcal{A}}$

$Add\ (\pi, i_B)\ to\ e_{\mathcal{F}}$

$Add\ (i_B, \kappa)\ to\ d_{\mathcal{G}}$

$end\ for$

$Let\ B = Out_{\kappa}.$

$For\ each\ (\kappa, \pi) \in d_{\mathcal{A}}$

$Add\ (o_B, \pi)\ to\ d_{\mathcal{F}}$

$Add\ (\kappa, o_B)\ to\ e_{\mathcal{G}}$

$end\ for$

$end\ if$

$end\ for$

Theorem 8.23. *Let $\mathcal{A}$ be a precode, and let $\mathcal{F}$ and $\mathcal{G}$ be as defined in Algorithm 8.22. Then $(\mathcal{F}, \mathcal{G})$ is a minimal parametrization of $\mathcal{A}$.*

Proof. We need only show that $(\mathcal{F}, \mathcal{G})$ is a parametrization of $\mathcal{A}$ since minimality follows from the discussion preceding Algorithm 8.22. However, the proof that $(\mathcal{F}, \mathcal{G})$ is a parametrization is similar to those given in the earlier theorems of this section. $\square$

Definition 8.24. *Parameterizations $(\mathcal{F}, \mathcal{G})$ and $(\mathcal{F}', \mathcal{G}')$ are said to be isomorphic if $\mathcal{F}$ is isomorphic to $\mathcal{F}'$ and $\mathcal{G}$ is isomorphic to $\mathcal{G}'$.*

We now compare the parametrization in Algorithm 8.22 with the earlier ones by revisiting Examples 8.15, 8.16, 8.17, and 8.18.

Example 8.25. *Let $\mathcal{A} = (P_{\mathcal{A}} = \{1\}, C_{\mathcal{A}} = \{a\}, e_{\mathcal{A}} = \{(1,a)\}, d_{\mathcal{A}} = e_{\mathcal{A}}^{nv})$ as in Example 8.15. The parametrization $(\mathcal{F}'', \mathcal{G}'')$ of $\mathcal{A}$ given in Algorithm 8.22 satisfies $\mathcal{F}'' = \mathcal{A}$ and*

$$\mathcal{G}'' = (P_{\mathcal{G}''} = C_{\mathcal{A}}, C_{\mathcal{G}''} = C_{\mathcal{A}}, e_{\mathcal{G}''} = \{(a,a)\}, d_{\mathcal{G}''} = e_{\mathcal{G}''}^{nv}).$$

Note that $(\mathcal{F}'', \mathcal{G}'')$ is the parametrization $(\mathcal{F}, \mathcal{G})$ of $\mathcal{A}$ given in Algorithm 8.5.

Example 8.26. *Let $\mathcal{A} = (P_{\mathcal{A}} = \{1\}, C_{\mathcal{A}} = \{a,b\}, e_{\mathcal{A}} = \{(1,a),(1,b)\}, d_{\mathcal{A}} = \{(a,1)\})$ as in Example 8.16. The parametrization $(\mathcal{F}'', \mathcal{G}'')$ of $\mathcal{A}$ given in Algorithm 8.22 is the parametrization $(\mathcal{F}', \mathcal{G}')$ given in Algorithm 8.12.*

Example 8.27. *Let $\mathcal{A} = (P_{\mathcal{A}} = \{1\}, C_{\mathcal{A}} = \{a,b,c\}, e_{\mathcal{A}} = \{(1,a),(1,b),(1,c)\}, d_{\mathcal{A}} = e_{\mathcal{A}}^{nv})$ as in Example 8.17. The parametrization $(\mathcal{F}'', \mathcal{G}'')$ of $\mathcal{A}$ given in Algorithm 8.22 is the parametrization $(\mathcal{F}', \mathcal{G}')$ given in Algorithm 8.12.*

Example 8.28. *Let $\mathcal{A} = (P_{\mathcal{A}} = \{1,2\}, C_{\mathcal{A}} = \{a,b,c\}, e_{\mathcal{A}} = \{(1,a),(1,b),(2,c)\}, d_{\mathcal{A}} = e_{\mathcal{A}}^{nv})$ as in Example 8.18. Also as in Example 8.18, let $(\mathcal{F}, \mathcal{G})$ be the parametrization of $\mathcal{A}$ given by Algorithm 8.5, and let $(\mathcal{F}', \mathcal{G}')$ be the parametrization of $\mathcal{A}$ given by Algorithm 8.12.*

The parametrization $(\mathcal{F}'', \mathcal{G}'')$ of $\mathcal{A}$ given in Algorithm 8.22 satisfies $e_{\mathcal{F}''} = \{(1, i_{\{1\}}), (2,c)\}$, $d_{\mathcal{F}''} = \{(o_{\{1\}}, 1), (c, 2)\}$, $e_{\mathcal{G}''} = \{(a, o_{\{1\}}), (b, o_{\{1\}}), (c,c)\}$, and $d_{\mathcal{G}''} = \{(i_{\{1\}}, a), (i_{\{1\}}, b), (c,c)\}$. See Figure 28.

114
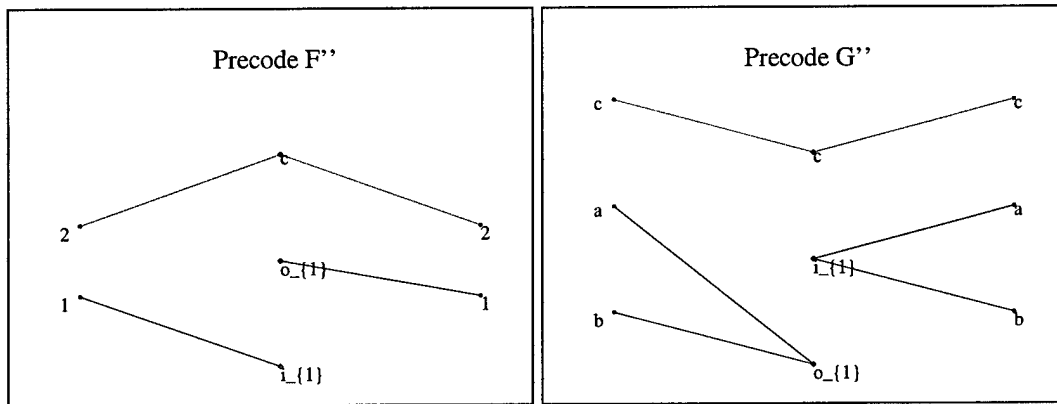
FIGURE 28. The Parametrization Given by Algorithm 8.22 (Example 8.28)

Note that $\mathcal{G}$, $\mathcal{G}'$, and $\mathcal{G}''$ each have 6 edges. This example is interesting since $\mathcal{F}''$ and $\mathcal{F}'$ have the same number of edges, but $\mathcal{F}''$ has fewer codetext elements than $\mathcal{F}'$. Also, $\mathcal{F}''$ and $\mathcal{F}$ have the same number of codetext elements, but $\mathcal{F}''$ has fewer edges than $\mathcal{F}$. Thus, $(\mathcal{F}'', \mathcal{G}'')$ is an improvement over both $(\mathcal{F}, \mathcal{G})$ and $(\mathcal{F}', \mathcal{G}')$.

9. CONCLUSION

The purpose of this paper was to further the work begun in [2], [3], and [4]. In particular, we accomplished the following tasks:

(1) Rewrote [2] and [3] in terms of the alternate representations of precodes as bipartite digraphs and Boolean matrices.

(2) Counted various types of bipartite graphs up to isomorphism, and counted various classes of codes and precodes up to isomorphism.

(3) Identified many of the classical objects and morphisms from category theory within the categories $\mathfrak{P}$ and $\mathfrak{C}$.

(4) Described various ways of constructing a code from a precode by "splitting" the precode, identifying important properties of these constructions and their interrelationship. Discussed the properties of the constructed codes with regard to the factorization of homomorphisms through them, and discussed their relationship to the code constructed from the precode by "smashing."

(5) Defined a parametrization of a precode and gave constructions of various parametrizations of a given precode, including a "minimal" parametrization.

(6) Used the computer algebra system, Maple, to represent and display a precode and its opposite, smash, split, bald-split, and various parametrizations. Implemented the formulae developed for counting bipartite graphs and precodes up to isomorphism.

Although many topics were explored extensively, there are still promising possibilities for future research within the "general theory of codes." For example, it appears that precodes might be used to describe general access structures as described in [9]. In this case, Shamir's secret-sharing scheme (see [10]) is immediately described in terms of precodes.

REFERENCES

[1] G. R. Blakley and I. Borosh, Codes, unpublished manuscript, 1995.

[2] ———, A general theory of codes i: Basic concepts, in "Proceedings of the Klagenfurt Conference, May 29-June 1, 1997," Contributions to General Algebra, Vol. 10, Klagenfurt, Austria, (1998), 1–29. MR **99m**:94062

[3] ———, A general theory of codes ii: Paradigms and homomorphisms, in "First International Workshop, Proceedings/ISW'97, September 17-19, 1997," Information Security, Vol. 1396, Tatsunokuchi, Ishikawa, Japan, (1998), 1–30.

[4] G. R. Blakley, I. Borosh, T. Holcomb, and A. Klappenecker, Categorical code constructions, in preparation.

[5] F. Harary and E. M. Palmer, "Graphical enumeration," Academic Press, New York, 1973. MR **50** #9682

[6] H. Herrlich and G. E. Strecker, "Category theory," Allyn and Bacon, Inc., Boston, 1973. MR **50** #2284

[7] T. W. Hungerford, "Algebra," Springer-Verlag, New York, 1974. MR **82a**:00006

[8] S. Mac Lane, "Categories for the working mathematician," Springer-Verlag, New York, 1998. MR **2001j**:18001

[9] A. J. Menezes, P. van Oorschot, and S. Vanstone, "Handbook of applied cryptography," CRC Press, Boca Raton, FL, 1997. MR **99g**:94015

[10] A. Shamir, How to share a secret, *Communications of the ACM* **22** (1979), no. 11, 612–613. MR **80g**:94070

APPENDIX A

PRECODE PRELIMINARIES

A.1. **Precodes as Strip Charts.** As documented in $6A$ in [2], a two-strip chart is a useful way to present precodes graphically. A strip chart is composed of vertical axes alternating with vertical strips. A vertical axis consists of symbols representing the elements in some set. A vertical strip consists of line segments between points in two vertical axes, representing a relation between the sets represented by the axes. To display a precode $\mathcal{A}$, we use the format $PeCdP$, where P denotes a vertical axis representing $P_\mathcal{A}$, C is a vertical axis representing $C_\mathcal{A}$, e is a vertical strip representing $e_\mathcal{A}$, and d is a vertical strip representing $d_\mathcal{A}$. We give the following example to demonstrate the above discourse.

Example A.1. *Let*

$$\mathcal{A} = (P_\mathcal{A} = \{p_1\}, C_\mathcal{A} = \{c_1, c_2, c_3\}, e_\mathcal{A} = \{(p_1, c_2), (p_2, c_1), (p_2, c_2)\}, d_\mathcal{A} = \{(c_1, p_2), (c_3, p_2)\}.$$

Then a strip-chart presentation of $\mathcal{A}$ is depicted below, where we use $p1$ to represent p_1, etc.
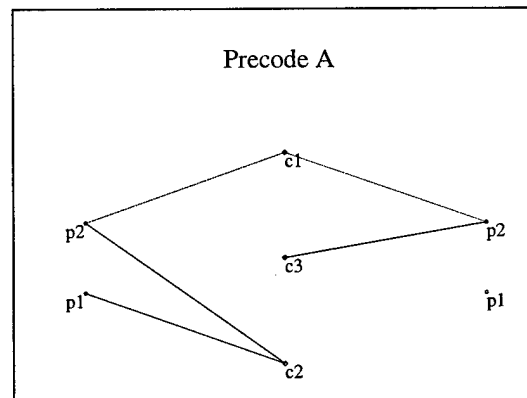
FIGURE 29. A Strip Chart Representation of a Precode

In addition to the set-theoretic definition of a precode given above, there are also useful representations of precodes in terms of bipartite digraphs and matrices.

A.2. Precodes as Bipartite Digraphs. As in section $5A$ of [2], we show how bipartite graphs can represent precodes and vice versa. Let $P_\mathcal{A} \cup C_\mathcal{A}$ be the vertex set of a bipartite digraph $\mathcal{A}$, where $P_\mathcal{A}$ and $C_\mathcal{A}$ are disjoint sets. Let $e_\mathcal{A}$ be a set of directed edges from $P_\mathcal{A}$ to $C_\mathcal{A}$; that is, let $e_\mathcal{A} \subseteq P_\mathcal{A} \times C_\mathcal{A}$. Also, let $d_\mathcal{A} \subseteq C_\mathcal{A} \times P_\mathcal{A}$ be a set of directed edges from $C_\mathcal{A}$ to $P_\mathcal{A}$. We can represent $\mathcal{A}$ by the four-tuple, $\mathcal{A} = (P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A}, d_\mathcal{A})$. Hence, we can view $\mathcal{A}$ as a precode.

Similarly, we can represent a precode $\mathcal{A} = (P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A}, d_\mathcal{A})$ by a digraph. The sets $P_\mathcal{A}$ and $C_\mathcal{A}$ can be viewed as disjoint vertex sets (by relabeling if necessary), and the relations $e_\mathcal{A} \subseteq P_\mathcal{A} \times C_\mathcal{A}$ and $d_\mathcal{A} \subseteq C_\mathcal{A} \times P_\mathcal{A}$ specify adjacency structures between these sets. The bipartite digraph associated with $\mathcal{A}$ is the graph whose vertex set is $P_\mathcal{A} \cup C_\mathcal{A}$ and whose edge set is given by $e_\mathcal{A} \cup d_\mathcal{A}$.

With a slight abuse of notation, we will call the ordered pairs in $e_\mathcal{A} \cup d_\mathcal{A}$ edges in $\mathcal{A}$. As in Theorem $5A1$ in [2], we note that if $\mathcal{A}$ is a code, then the bipartite graph representing $\mathcal{A}$ has the property that every two-arc path beginning in $P_\mathcal{A}$ (and thus also ending in $P_\mathcal{A}$) is a circuit. Theorem $5A2$ states that every bipartite digraph $\mathcal{A} = (P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A}, d_\mathcal{A})$ with the property that every two-arc path which begins (and thus ends) in $P_\mathcal{A}$ is a circuit represents a code from $P_\mathcal{A}$ to $C_\mathcal{A}$.

A.3. Precodes as Boolean Matrices. The following comes from the discussion in Section 5 of [1]. The zero-one complete atomic Boolean algebra $ZOBAP = (\{0,1\}, \vee, \wedge, \neg, \leq)$ satisfies

$$0 \wedge 0 = 0 \wedge 1 = 1 \wedge 0 = 0 \vee 0 = \neg 1 = 0,$$

$$1 \vee 0 = 0 \vee 1 = 1 \vee 1 = 1 \wedge 1 = \neg 0 = 1.$$

We note that it is also a poset (partially ordered set) in which $0 \leq 1$.

Let $o = 00, d = 01, e = 10$, and $s = 11$. The synoptic complete atomic Boolean algebra $SYBAP = (\{o, d, e, s\}, \vee, \wedge, \neg, \leq)$ is defined via

$$o \wedge e = o \wedge d = o \wedge s = e \wedge d = \neg s = o,$$

$$o \vee d = d \vee d = d \wedge d = d \wedge s = \neg e = d,$$

$$o \vee e = e \vee e = e \wedge e = e \wedge s = \neg d = e,$$

$$s \vee o = s \vee e = s \vee d = e \vee d = \neg o = s.$$

$SYBAP$ is a poset satisfying $o \leq e \leq s$ and $o \leq d \leq s$.

We can view each of o, d, e, and s as two-vectors over $ZOBAP$. In this case, the $SYBAP$ operations $\vee$, $\wedge$, $\neg$, $\leq$ are precisely the corresponding operations in $ZOBAP$ applied coordinate-wise.

The following corresponds to Definition $5B.1$ in [1].

Definition A.2. *Let $\mathcal{A}$ be a precode. The synoptic codebook matrix $M_\mathcal{A}$ of $\mathcal{A}$ is defined by setting*

$$M_\mathcal{A}(\pi, \kappa) = \begin{cases} s \text{ if } (\pi, \kappa) \in e_\mathcal{A} \text{ and } (\kappa, \pi) \in d_\mathcal{A}, \\ e \text{ if } (\pi, \kappa) \in e_\mathcal{A} \text{ and } (\kappa, \pi) \notin d_\mathcal{A}, \\ d \text{ if } (\pi, \kappa) \notin e_\mathcal{A} \text{ and } (\kappa, \pi) \in d_\mathcal{A}, \\ o \text{ if } (\pi, \kappa) \notin e_\mathcal{A} \text{ and } (\kappa, \pi) \notin d_\mathcal{A}. \end{cases}$$

We can view the entries of $M_\mathcal{A}$ as two-bit vectors and reference them accordingly. The left bit is the encode (or E) bit, and the right bit is the decode (or D) bit. For example, if $M_\mathcal{A}(\pi, \kappa) = e = 10$, we may write $M_\mathcal{A}(\pi, \kappa)(E) = 1$ and $M_\mathcal{A}(\pi, \kappa)(D) = 0$.

Notice that each column in $M_\mathcal{A}$ represents the local structure of $\mathcal{A}$ with respect to a particular codetext element.

A.4. Subprecodes, Unions, and Intersections. The following is a restatement of $2D1$ in [2].

Definition A.3. *Let $\mathcal{A}$ and $\hat{\mathcal{A}}$ be precodes. $\hat{\mathcal{A}}$ is a subprecode of $\mathcal{A}$ (and $\mathcal{A}$ is a superprecode of $\hat{\mathcal{A}}$) if $P_{\hat{\mathcal{A}}} \subseteq P_\mathcal{A}$, $C_{\hat{\mathcal{A}}} \subseteq C_\mathcal{A}$, $e_{\hat{\mathcal{A}}} \subseteq e_\mathcal{A}$, and $d_{\hat{\mathcal{A}}} \subseteq d_\mathcal{A}$. A subprecode which is a code is called a subcode.*

Similarly, a superprecode which is a code is a supercode. We note that a subprecode of a code is a code.

The following comes from $2D2$ in [2].

Definition A.4. *The intersection of the precodes $\mathcal{A}$ and $\hat{\mathcal{A}}$ is the precode*

$$\mathcal{A} \cap \hat{\mathcal{A}} = (P_\mathcal{A} \cap P_{\hat{\mathcal{A}}}, C_\mathcal{A} \cap C_{\hat{\mathcal{A}}}, e_\mathcal{A} \cap e_{\hat{\mathcal{A}}}, d_\mathcal{A} \cap d_{\hat{\mathcal{A}}}).$$

We note that $\mathcal{A} \cap \hat{\mathcal{A}}$ is a code if either $\mathcal{A}$ or $\hat{\mathcal{A}}$ is a code.

A.5. Companions, Self-Companion Codes, and Nubs. The following is a restatement of $3A1$ in [2].

Definition A.5. *Let $\mathcal{A}$ be a precode. The companion $\mathcal{A}^{pn}$ of $\mathcal{A}$ is the precode $\mathcal{A}^{pn} = (P_A, C_A, d_A^{\leftarrow}, e_A^{\leftarrow})$. If $\mathcal{A} = \mathcal{A}^{pn}$, then $\mathcal{A}$ is called a self-companion precode. The companion of a code, and a self-companion code, are defined analogously.*

The following is Definition $3A8$ in [2].

Definition A.6. *Let $\mathcal{A}$ be a precode. The self-companion kernel (the nub) of $\mathcal{A}$ is the precode $N(\mathcal{A}) = (P_A, C_A, c, c^{\leftarrow})$, where c is as in Definition 2.2.*

A.6. Opposites, Self-Opposite Codes, and Hinges. The following is a restatement of $3B1$ in [2].

Definition A.7. *Let $\mathcal{A}$ be a precode. The the precode $\mathcal{A}^{op} = (C_A, P_A, d_A, e_A)$ is called the opposite of $\mathcal{A}$. If $\mathcal{A}^{op} = \mathcal{A}$, then $\mathcal{A}$ is said to be self-opposite. The opposite of a code, and a self-opposite code, are defined analogously.*

The following is a restatement of $3B6$ in [2].

Definition A.8. *Let $\mathcal{A}$ be a precode. The precode*

$$H = H(\mathcal{A}) = (P_A \cap C_A, P_A \cap C_A, e_A \cap d_A, e_A \cap d_A) = \mathcal{A} \cap \mathcal{A}^{op}$$

is called the hinge of $\mathcal{A}$.

A.7. Janiform Codes. The following is a restatement of $3C1$ in [2].

Definition A.9. *Let $\mathcal{A}$ be a precode. $\mathcal{A}$ is a janiform precode if $\mathcal{A}^{op}$ is a code. A janiform code is a code whose opposite is also a code.*

A.8. Self-Companion, Self-Opposite Codes. The following is a restatement of $3D4$ in [2].

Definition A.10. *Let $\mathcal{A}$ be a precode. We call $\mathcal{A}^{oppn} = (\mathcal{A}^{op})^{pn} = (\mathcal{A}^{pn})^{op}$ the companion opposite of $\mathcal{A}$. The list of four precodes $[\mathcal{A}, \mathcal{A}^{pn}, \mathcal{A}^{op}, \mathcal{A}^{oppn}]$ is called the quartet of $\mathcal{A}$.*

A.9. Nulls. The following is $4A7$ in [2].

Definition A.11. *Let $\mathcal{A}$ be a precode. The set $e_A NL$ of encode nulls, the set $d_A NL$ of decode nulls, the set $c_A NL$ of circulation nulls, and the set $s_A NL$ of simultaneous nulls are defined by setting*

$$e_A NL = C_A \backslash RAN(e_A),$$

$$d_A NL = C_A \backslash DOM(d_A),$$

$$c_A NL = C_A \backslash RAN(c_A),$$

$$s_A NL = e_A NL \cap d_A NL.$$

Similarly, the set $e_A VD$ of encode voids, the set $d_A VD$ of decode voids, the set $c_A VD$ of circulation voids, and the set $s_A VD$ of simultaneous voids are defined by setting

$$e_A VD = P_A \backslash DOM(e_A),$$

$$d_A VD = P_A \backslash RAN(d_A),$$

$$c_A VD = P_A \backslash DOM(c_A),$$

$$s_A VD = e_A VD \cap d_A VD.$$

A.10. **Homomorphisms.** We start with homomorphisms between relations.

A.10.1. *Relation Homomorphisms.* As in [3], we say that (G, H, r) is a relation to mean that r is a binary relation from G to H. The following definitions correspond to $7A1$, $7A2$, and $7A4$ in [3].

Definition A.12. *A relation homomorphism from the relation (G, H, r) to the relation $(\hat{G}, \hat{H}, m)$ is a pair (g, h) of functions $g : G \to \hat{G}$ and $h : H \to \hat{H}$ such that $h \circ r \circ g^{\leftarrow} \subseteq m$. We call the relation $Im((g, h)) = (g(G), h(H), h \circ r \circ g^{\leftarrow})$ the image of the homomorphism (g, h). A relation homomorphism (g, h) is called a relation epimorphism if g and h are surjections.*

Definition A.13. *A relation homomorphism (g, h) from (G, H, r) to $(\hat{G}, \hat{H}, m)$ is a strong relation homomorphism if $m \subseteq h \circ r \circ g^{\leftarrow}$.*

Definition A.14. *Suppose that $g : G \to \hat{G}$ and $h : H \to \hat{H}$ are functions. Suppose that (g, h) is a relation homomorphism from (G, H, r) to $(\hat{G}, \hat{H}, m)$, and that the function pair $(g^{\leftarrow}, h^{\leftarrow})$ is a relation homomorphism from $(\hat{G}, \hat{H}, m)$ to (G, H, r). Then (g, h) is said to be a relation isomorphism from (G, H, r) to $(\hat{G}, \hat{H}, m)$.*

A.10.2. *Quotients and Canonical Maps.* The following definition is a combination of Definitions $6C1$ and $6C2$ and Lemma $6C3$ in [3].

Definition A.15. *If (G, G, s) is an equivalence relation, then the quotient, G modulo s, is the set*

$$G/s = \{s(\{\gamma\}) \mid \gamma \in G\};$$

i.e., G/s is the partition of G induced by the equivalence relation s. Its members are called cells.

Any function $g : G \to \hat{G}$ effects a partition $\Psi[g] = \{g^{\leftarrow}(\{\hat{\gamma}\}) \mid \hat{\gamma} \in RAN(g)\}$, where the cells are the fibers of g. It is clear that $s = g^{\leftarrow} \circ g$ is the equivalence relation such that $G/s = \Psi[g]$.

Furthermore, an equivalence relation (G, G, s) determines a canonical function $f_s : G \to G/s$ such that $f_s = \{(\gamma, s(\{\gamma\})) \mid \gamma \in G\}$. It is clear that $f_s^{\leftarrow} \circ f_s = s$ and $f_s \circ f_s^{\leftarrow} = diag((G/s)^2) = i_{G/s}$.

The following definition comes from $6C5$ in [3].

Definition A.16. *Let (G, G, r) be a relation, and let (G, G, s) and (H, H, t) be equivalence relations. We define the quotient relation*

$$(G, H, r)/(s, t) = (G/s, H/t, r/(s, t))$$

where

$$\begin{aligned} r/(s, t) &= f_t \circ r \circ f_s^{\leftarrow} \\ &= \{(s(\{\gamma\}), t(\{\eta\})) \mid (\gamma, \eta) \in r\}. \end{aligned}$$

We recall $7A6$ from [3].

Definition A.17. *Let (g, h) be a relation homomorphism from (G, H, r) to $(\hat{G}, \hat{H}, m)$. Then the equivalence relation pair*

$$Ker((g, h)) = (g^{\leftarrow} \circ g, h^{\leftarrow} \circ h)$$

is called the kernel of the homomorphism (g, h).

A.10.3. *Isomorphism Theorems for Relations.* The following theorem is stated and proven in [3] as Theorem 7B1. It is an analogue to the first isomorphism theorem of group theory.

Theorem A.18. *Let (g, h) be a relation homomorphism from (G, H, r) to $(\hat{G}, \hat{H}, m)$ with kernel $(s, t) = (g^{\leftarrow} \circ g, h^{\leftarrow} \circ h)$. Let (f_s, f_t) be the canonical map pair from (G, H, r) onto $(G, H, r)/(s, t)$. Then*

1. The natural map pair $n = (g \circ f_s^{\leftarrow}, h \circ f_t^{\leftarrow})$ is a relation homomorphism from $(G, H, r)/(s, t)$ to $(\hat{G}, \hat{H}, m)$.

2. If (g, h) is a strong relation epimorphism, then $n = (g \circ f_s^{\leftarrow}, h \circ f_t^{\leftarrow})$ is a relation isomorphism from $(G, H, r)/(s, t)$ to $(\hat{G}, \hat{H}, m)$.

A.10.4. *Precode Homomorphisms.* We are now ready to define precode homomorphisms using relation homomorphisms. The following definitions come from 8A1, 8A2, 8A3, and 8A4 in [3].

Definition A.19. *Let $\mathcal{A}$ and $\hat{\mathcal{A}}$ be precodes. A pair (g, h) of functions $g : P \to P_{\hat{A}}$ and $h : C_A \to C_{\hat{A}}$ is a precode homomorphism if the following two conditions hold:*

1. (g, h) is a relation homomorphism from (P_A, C_A, e_A) to $(P_{\hat{A}}, C_{\hat{A}}, e_{\hat{A}})$.

2. (h, g) is a relation homomorphism from (C_A, P_A, d_A) to $(C_{\hat{A}}, P_{\hat{A}}, d_{\hat{A}})$.

By Definition A.12, these two conditions are equivalent to requiring that

$$h \circ e_A \circ g^{\leftarrow} \subseteq e_{\hat{A}} \text{ and } g \circ d_A \circ h^{\leftarrow} \subseteq d_{\hat{A}}.$$

If g and h are surjections, then (g, h) is called a precode epimorphism.

The precode $Im((g, h)) = (g(P), h(C), h \circ e_A \circ g^{\leftarrow}, g \circ d_A \circ h^{\leftarrow})$ is called the image of the precode homomorphism (g, h), and we may also denote it by $(g, h)(\mathcal{A})$.

For a precode $\mathcal{A}$, we will often use $1_{\mathcal{A}}$ to denote the function pair $(1_{P_A}, 1_{C_A})$. That is, for any superprecode $\mathcal{B}$ of $\mathcal{A}$, $1_{\mathcal{A}} : \mathcal{A} \longrightarrow \mathcal{B}$ represents the natural inclusion precode homomorphism.

Definition A.20. *Let $(g, h) : \mathcal{A} \longrightarrow \hat{\mathcal{A}}$ be a precode homomorphism such that $e_{\hat{A}} \subseteq h \circ e_A \circ g^{\leftarrow}$ and $d_{\hat{A}} \subseteq g \circ d_A \circ h^{\leftarrow}$. Then (g, h) is called a strong precode homomorphism from $\mathcal{A}$ to $\hat{\mathcal{A}}$. Note that by*

Definition A.13, this is equivalent to requiring that (g, h) be a strong relation homomorphism from $(P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A})$ to $(P_{\hat{\mathcal{A}}}, C_{\hat{\mathcal{A}}}, e_{\hat{\mathcal{A}}})$ and (h, g) be a strong relation homomorphism from $(C_\mathcal{A}, P_\mathcal{A}, d_\mathcal{A})$ to $(C_{\hat{\mathcal{A}}}, P_{\hat{\mathcal{A}}}, d_{\hat{\mathcal{A}}})$. If g and h are surjections, then we say that (g, h) is a strong precode epimorphism.

Definition A.21. *Let $(g, h) : \mathcal{A} \longrightarrow \hat{\mathcal{A}}$ be a precode homomorphism. If $(g^\leftarrow, h^\leftarrow)$ is a precode homomorphism from $\hat{\mathcal{A}}$ to $\mathcal{A}$, then (g, h) is a precode isomorphism. This is equivalent to requiring that (g, h) be a relation isomorphism from $(P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A})$ to $(P_{\hat{\mathcal{A}}}, C_{\hat{\mathcal{A}}}, e_{\hat{\mathcal{A}}})$ and (h, g) be a relation isomorphism from $(C_\mathcal{A}, P_\mathcal{A}, d_\mathcal{A})$ to $(C_{\hat{\mathcal{A}}}, P_{\hat{\mathcal{A}}}, d_{\hat{\mathcal{A}}})$.*

The following example from [4] shows that not every precode homomorphism (g, h) for which g and h are bijections is a precode isomorphism. It is an extension of Example 2.28. However, we note that a strong precode homomorphism (g, h) for which g and h are bijections is a precode isomorphism.

Example A.22. *Let $G = \{0, 1\}$, and let $1_G : G \longrightarrow G$ be the identity function. Note that 1_G is a bijection, but $(1_G, 1_G) : (G, G, \emptyset, \emptyset) \longrightarrow (G, G, G \times G, G \times G)$ is a precode homomorphism which is not an isomorphism.*

A.10.5. *Isomorphism Theorems for Precodes.* This definition is the analogue of Definition A.17 for precodes.

Definition A.23. *Let $(g, h) : \mathcal{A} \longrightarrow \hat{\mathcal{A}}$ be a precode homomorphism. The equivalence relation pair*

$$Ker((g, h)) = (s, t) = (g^\leftarrow \circ g, h^\leftarrow \circ h)$$

is called the kernel of the homomorphism (g, h). We note that by the definition of a precode homomorphism, (g, h) is relation homomorphism from $(P_\mathcal{A}, C_\mathcal{A}, e_\mathcal{A})$ to $(P_{\hat{\mathcal{A}}}, C_{\hat{\mathcal{A}}}, e_{\hat{\mathcal{A}}})$ with kernel (s, t) and (h, g) is a relation homomorphism from $(C_\mathcal{A}, P_\mathcal{A}, d_\mathcal{A})$ to $(C_{\hat{\mathcal{A}}}, P_{\hat{\mathcal{A}}}, d_{\hat{\mathcal{A}}})$ with kernel (t, s).

A.11. **Products and Sums.** As in [4], we have the following definition.

Definition A.24. *Let $\mathcal{A}_i = (P_i, C_i, e_i, d_i)$ be a family of precodes indexed by a set I. The precode $\mathcal{A} = (\prod_{i\in I} P_i, \prod_{i\in I} C_i, \prod_{i\in I} e_i, \prod_{i\in I} d_i)$ is called the product of the $\mathcal{A}_i$.*

As in the proof of Theorem 12 in [4], we have the following definition.

Definition A.25. *Let $\{\mathcal{A}_i = (P_i, C_i, e_i, d_i) \mid i \in I\}$ be a family of precodes indexed by a set I. The precode $\mathcal{A} = (\bigcup_{i\in I} P_i \times \{i\}, \bigcup_{i\in I} C_i \times \{i\}, \bigcup_{i\in I} e_i \times \Delta_i, \bigcup_{i\in I} d_i \times \Delta_i)$ is called the direct sum of the $\mathcal{A}_i$, where Δ_i denotes the relation $\Delta_i = \{(i,i)\}$.*

A.12. **The Smash of a Precode.** The following comes from [4].

Definition A.26. *Let $\mathcal{A}$ be a precode. Let E denote the smallest equivalence relation on $P_\mathcal{A}$ containing $d_\mathcal{A} \circ e_\mathcal{A}$, and let I denote the identity relation on $C_\mathcal{A}$. The code*

$$\mathcal{A}_\# = (P_\mathcal{A}/E, C_\mathcal{A}, e_\mathcal{A}/(E,I), d_\mathcal{A}/(I,E))$$

*is called the **smash of** $\mathcal{A}$.*

APPENDIX B

CATEGORY THEORY PRELIMINARIES

B.1. Categories. The following is a combination of Definitions 3.1 and 4.12 in [6].

Definition B.1. *A category is a quintuple* $\mathcal{C} = (O, M, dom, cod, \circ)$ *where*

(i) *O is a class whose members are called $\mathcal{C}$ objects,*

(ii) *M is a class whose members are called $\mathcal{C}$ morphisms*

(iii) *dom and cod are functions from M to O ($dom(f)$ is called the domain of f, and $cod(f)$*

is called the codomain of f),

(iv) *$\circ$ is a function from $D = \{(f, g) \mid f, g \in M$ and $dom(f) = cod(g)\}$ into M, called the*

composition law of $\mathcal{C}$ ($\circ(f, g)$ is usually written $f \circ g$ and we say that $f \circ g$ is defined

if and only if $(f, g) \in D$);

such that the following conditions are satisfied:

(1) *Matching Condition: If $f \circ g$ is defined, then $dom(f \circ g) = dom(g)$ and $cod(f \circ g) = cod(f)$;*

(2) *Associativity Condition: If $f \circ g$ and $h \circ f$ are defined, then $h \circ (f \circ g) = (h \circ f) \circ g$;*

(3) *Identity Existence Condition: For each $\mathcal{C}$-object A there exists a $\mathcal{C}$-morphism 1_A such that*

$dom(1_A) = A = cod(1_A)$ and

(a) *$f \circ 1_A = f$ whenever $f \circ 1_A$ is defined, and*

(b) *$1_A \circ g = 1_A$ whenever $1_A \circ g$ is defined;*

(4) *Smallness of the Morphism Class Condition: For any pair (A, B) of $\mathcal{C}$-objects, the class*

$hom_{\mathcal{C}}(A, B) = \{f \mid f \in M, dom(f) = A,$ and $cod(f) = B\}$ is a set.

The opposite (or dual) category of $\mathcal{C}$ is the category $\mathcal{C}^{op} = \mathcal{C} = (O, M, cod, dom, *)$, where $*$ is

defined by $f * g = g \circ f$.

Notation B.2. *Let $\mathcal{C}$ be a category. For a given $\mathcal{C}$-object A, the morphism 1_A satisfying 3(a) and*

3(b) is unique. We will typically denote the class of $\mathcal{C}$-objects by $Ob(\mathcal{C})$ and the class of $\mathcal{C}$-morphisms

by $Mor(\mathcal{C})$. We will often use $hom(A, B)$ to denote $hom_{\mathcal{C}}(A, B)$ when no confusion will arise.

B.2. Subcategories. We begin with Definition 4.1 in[6].

Definition B.3. *A category* $\mathcal{B}$ *is said to be a subcategory of the category* $\mathcal{C}$ *provided that the following conditions hold:*

(1) $Ob(\mathcal{B}) \subseteq Ob(\mathcal{C})$.

(2) $Mor(\mathcal{B}) \subseteq Mor(\mathcal{C})$.

(3) The domain, codomain and composition functions of $\mathcal{B}$ *are restrictions of the corresponding functions of* $\mathcal{C}$.

(4) Every $\mathcal{B}$-*identity is a* $\mathcal{C}$-*identity.*

Note that (2) and (3) imply that $hom_{\mathcal{B}}(A, B) \subseteq hom_{\mathcal{C}}(A, B)$

Definition B.4. *A subcategory* $\mathcal{B}$ *of a category* $\mathcal{C}$ *is a full subcategory of* $\mathcal{C}$ *provided that for all* $A, B \in Ob(\mathcal{B})$, $hom_{\mathcal{B}}(A, B) = hom_{\mathcal{C}}(A, B)$.

As in [4], we have the following definition.

Definition B.5. *The category of precodes,* $\mathfrak{P}$, *is the category which has precodes as objects and precode homomorphisms as morphisms. The composition of arrows is given by the composition of functions. The category of codes,* $\mathfrak{C}$, *has codes as objects and precode homomorphisms as morphisms.*

It is clear that $\mathfrak{C}$ is a full subcategory of $\mathfrak{P}$.

B.3. Morphisms. The following is a combination of Definitions 6.2 and 6.9 in [6].

Definition B.6. *Let* $\mathcal{C}$ *be a category. A* $\mathcal{C}$-morphism $A \xrightarrow{f} B$ *is said to be a monomorphism in* $\mathcal{C}$ *provided that for all* $\mathcal{C}$-morphisms h and k *such that* $f \circ h = f \circ k$, *it follows that* $h = k$. *The morphism* f *is said to be an epimorphism in* $\mathcal{C}$ *if for all* $\mathcal{C}$-morphisms h and k *such that* $h \circ f = k \circ f$, *it follows that* $h = k$.

The following is Proposition 1 from [4].

Proposition B.7. *Let $f = (f_1, f_2) : A \to B$ be a precode homomorphism.*

(a) f is a monomorphism if and only if f_1 and f_2 are injective functions.

(b) f is an epimorphism if and only if f_1 and f_2 are surjective functions.

The following is a combination of Definitions 5.2, 5.7, 5.13, and 6.16 in [6].

Definition B.8. *Let $\mathcal{C}$ be a category. A $\mathcal{C}$-morphism $A \xrightarrow{f} B$ is said to be a section (resp. retraction) in $\mathcal{C}$ if there exists some $\mathcal{C}$-morphism $B \xrightarrow{g} A$ such that $g \circ f = 1_A$ (resp. $f \circ g = 1_A$). It is an isomorphism in $\mathcal{C}$ if it is both a $\mathcal{C}$-section and a $\mathcal{C}$-retraction. It is a bimorphism in $\mathcal{C}$ if it is both a monomorphism and an epimorphism. $\mathcal{C}$ is said to be balanced if each of its bimorphisms is an isomorphism.*

The following is a combination of Propositions 5.4, 5.10, 5.16, 6.4, 6.12, and 6.20 in [6].

Proposition B.9. *If $A \xrightarrow{f} B$ and $B \xrightarrow{g} C$ are $\mathcal{C}$-sections (resp. retractions, isomorphisms, monomorphisms, epimorphisms, or bimorphisms), then $A \xrightarrow{g \circ f} C$ is a $\mathcal{C}$-section (resp. retraction, isomorphism, monomorphism, epimorphism, or bimorphism).*

We note that the categorical notion of isomorphism in the categories $\mathfrak{P}$ and $\mathfrak{C}$ coincides with the definition given in Definition A.21. We also note that if $f = (f_1, f_2)$ is an isomorphism in $\mathfrak{P}$ or $\mathfrak{C}$, then f_1 and f_2 are bijections.

Proposition B.10. *The categories $\mathfrak{P}$ and $\mathfrak{C}$ are not balanced.*

Proof. See Example A.22. $\square$

B.4. **Subobjects.** The following comes from Definitions 6.22 and 6.23 in [6].

Definition B.11. *A subobject of an object B in a category $\mathcal{C}$ is a pair (A, f) where $A \xrightarrow{f} B$ is a monomorphism. Dually, (f, A) is a quotient object of B if $B \xrightarrow{f} A$ is an epimorphism.*

If (A, f) and (C, g) are subobjects of B, then (A, f) is said to be smaller than (C, g) (denoted by $(A, f) \leq (C, g)$) if and only if there exists some morphism $A \xrightarrow{h} C$ such that the triangle

$$
\begin{array}{ccc}
A & & \\
\vdots & \searrow{f} & \\
h & & B \\
\vdots & \nearrow{g} & \\
C & &
\end{array}
$$
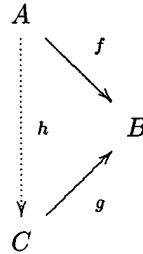
commutes; i.e., $g \circ h = f$. If $(A, f) \leq (C, g)$ and $(C, g) \leq (A, f)$, then (A, f) and (C, g) are said to be isomorphic subobjects of B, denoted by $(A, f) \approx (C, g)$.

Remark B.12. *By Proposition 6.24 in [6], subobjects (A, f) and (C, g) of B are isomorphic subobjects of B if and only if there is a unique isomorphism $A \xrightarrow{h} C$ such that $g \circ h = f$. The class of all subobjects of an object B is partitioned into equivalence classes of isomorphic subobjects.*

B.5. Well-powered and Co-(well-powered). The following comes from Definition 6.27 in [6].

Definition B.13. *A category $\mathcal{C}$ is said to be well-powered provided that each $\mathcal{C}$-object has a representative class of subobjects that is a set. Dually, $\mathcal{C}$ is said to be co-(well-powered) provided that each $\mathcal{C}$-object has a representative class of quotient objects which is a set.*

B.6. Intersections. The following comes from Definitions 17.2 and 17.5 in [6].

Definition B.14. *If B is a $\mathcal{C}$-object and $(A_i, m_i)_{i \in I}$ is a family of subobjects of B, then the pair (D, d) is called an intersection in $\mathcal{C}$ of $(A_i, m_i)_{i \in I}$ provided that*

(1) $d : D \longrightarrow B$ is a $\mathcal{C}$-monomorphism;

(2) for each $i \in I$ there is a $\mathcal{C}$-monomorphism $d_i : D \longrightarrow A_i$ with the property that $m_i \circ d_i = d$;

(3) if $g : C \longrightarrow B$ and for each $i \in I$, $g_i : C \longrightarrow A_i$ such that $m_i \circ g_i = g$, then there exists a unique $\mathcal{C}$-morphism $f : C \longrightarrow D$ such that the triangle

$$
\begin{array}{ccc}
& C & \\
f \downarrow & \searrow{\scriptstyle g} & \\
D & \xrightarrow[d]{} & B
\end{array}
$$

commutes; i.e., $d \circ f = g$.

$\mathcal{C}$ is said to have intersections if every set-indexed family of subobjects of each $\mathcal{C}$-object has an intersection.

The following is Proposition 17.3 in [6].

Proposition B.15. *Every intersection (D, d) of a family of subobjects $(A_i, m_i)_{i \in I}$ of an object B is a subobject of B; i.e., d is a monomorphism. Furthermore, (D, d) is (up to isomorphism) the largest subobject (relative to the order $\leq$ on subobjects) that is smaller than each of the subobjects $(A_i, m_i)_{i \in I}$.*

B.7. Products and Coproducts. The following comes from Definition 18.5 in [6].

Definition B.16. *A $\mathcal{C}$-product of a family $(A_i)_{i \in I}$ of $\mathcal{C}$-objects is a pair $(\prod_{i \in I} A_i, \prod_{i \in I} \pi_i)$ where $\prod_{i \in I} A_i$ is a $\mathcal{C}$-object and $\pi_i : \prod_{i \in I} A_i \longrightarrow A$ are $\mathcal{C}$-morphisms (called projections) with the property that if C is any $\mathcal{C}$-object and $f_i : C \longrightarrow A$ are arbitrary $\mathcal{C}$-morphisms, then there exists a unique $\mathcal{C}$-morphism $< f_i >: C \longrightarrow \prod_{i \in I} A_i$ such that for each $j \in I$, the diagram*

$$
\begin{array}{ccc}
C & \xrightarrow{<f_i>} & \prod_{i \in I} A_i \\
& \searrow{\scriptstyle f_j} & \downarrow{\scriptstyle \pi_j} \\
& & A
\end{array}
$$

commutes. We will often denote $(\prod_{i \in I} A_i, \prod_{i \in I} \pi_i)$ by $\prod_{i \in I} A_i$.

A $\mathcal{C}$-coproduct of the family $(A_i)_{i \in I}$ is a pair $(\coprod_{i \in I} \mu_i, \coprod_{i \in I} A_i)$ where $\coprod_{i \in I} A_i$ is a $\mathcal{C}$-object and $\mu_i : A \longrightarrow \coprod_{i \in I} A_i$ are $\mathcal{C}$-morphisms (called injections) with the property that if C is any

C-object and $f_i : A \longrightarrow C$ are arbitrary C-morphisms, then there exists a unique C-morphism $[f_i] : \coprod_{i \in I} A_i \longrightarrow C$ making the diagram

$$
\begin{array}{ccc}
& A & \\
\mu_j \downarrow & & \searrow f_j \\
\coprod_{i \in I} A_i & \xrightarrow[{[f_i]}]{} & C
\end{array}
$$

commute for each $j \in I$. We will often denote $(\coprod_{i \in I} \mu_i, \coprod_{i \in I} A_i)$ by $\coprod_{i \in I} A_i$.

The following is a combination of Theorems 9 and 12 in [4].

Theorem B.17. *The categories $\mathfrak{P}$ and $\mathfrak{C}$ have products and coproducts.*

We note that the product as defined in A.24 is such a product, and the direct sum as defined in A.25 is the corresponding coproduct.

B.8. Equalizers and Coequalizers. We have the following definition from Definition 16.2 in [6].

Definition B.18. *Let $A \xrightarrow{f} B$ and $A \xrightarrow{g} B$ be a pair of C-morphisms. A pair (E, e) is called an equalizer in C of f and g provided that the following conditions hold:*

(1) $e : E \longrightarrow A$ is a C-morphism;

(2) $f \circ e = g \circ e$;

(3) For any C-morphism $e' : E' \longrightarrow A$ such that $f \circ e' = g \circ e'$, there exists a unique C-morphism $\bar{e} : E' \longrightarrow E$ making the triangle

$$
\begin{array}{ccc}
& E' & \\
\bar{e} \downarrow & & \searrow e' \\
E & \xrightarrow[e]{} & A
\end{array}
$$

commute. Dually, if $c : B \to C$, *then* (c, C) *is called the coequalizer in* $\mathcal{C}$ *of the morphisms* $A \overset{f}{\longrightarrow} B$ *and* $A \overset{g}{\longrightarrow} B$ *if and only if* $c \circ f = c \circ g$ *and each* $\mathcal{C}$*-morphism* $c' : E' \longrightarrow A$ *satisfying* $c' \circ f = c' \circ g$ *can be uniquely factored through* c.

Remark B.19. *By Proposition 16.5 in* [6], *any two equalizers of a given pair of morphisms* $A \overset{f}{\longrightarrow} B$ *and* $A \overset{g}{\longrightarrow} B$ *are isomorphic subobjects of* A. *For this reason, we refer to* **the** *equalizer of* f *and* g *and denote it by* $Equ(f, g)$. *We similarly use* $Coeq(f, g)$ *to denote the coequalizer of morphisms* f *and* g.

The following is a combination of Theorems 10, 13, and 14 in [4].

Theorem B.20. *The categories* $\mathfrak{P}$ *and* $\mathfrak{C}$ *have equalizers and coequalizers.*

The following was shown in [4].

Lemma B.21. *Let* $f = (f_1, f_2), g = (g_1, g_2) : \mathcal{A} \longrightarrow (P, C, e, d)$ *be precode homomorphisms. The equalizer of* f *and* g *is* (P^*, C^*, e^*, d^*), *where*

$$P^* = \{\alpha \in P \mid f_1(\alpha) = g_1(\alpha)\}, C^* = \{\beta \in C \mid f_2(\beta) = g_2(\beta)\}, e^* = e_A|_{P^* \times C^*}, \text{ and } d^* = d_A|_{C^* \times P^*}.$$

B.9. Regular Morphisms. The following comes from Definition 16.13 in [6].

Definition B.22. *If* $H \overset{h}{\longrightarrow} A$ *is a* $\mathcal{C}$*-morphism, then* (H, h) *is called a regular subobject of* A *and* h *is called a regular monomorphism if and only if there are* $\mathcal{C}$*-morphisms* f *and* g *such that* $(H, h) \approx Equ(f, g)$.

Dually, if $B \overset{h}{\longrightarrow} H$ *is a* $\mathcal{C}$*-morphism, then* (h, H) *is called a regular quotient object of* $\mathcal{B}$ *and* h *is called a regular epimorphism if and only if there are* $\mathcal{C}$*-morphisms* f *and* g *such that* $(h, H) \approx Coeq(f, g)$.

By Proposition 16.15 in [6] *and duality, regular monomorphisms are monomorphisms, and regular epimorphisms are epimorphisms.*

B.10. Extremal Morphisms. The following comes from Definition 17.9 in [6].

Definition B.23. *An epimorphism e is called an extremal epimorphism provided that if $e = m \circ f$, where m is a monomorphism, then m must be an isomorphism. If $A \xrightarrow{e} B$ is an extremal epimorphism, then (e, B) is called an extremal quotient object of A.*

A monomorphism m is called an extremal monomorphism provided that if $m = f \circ e$, where e is an epimorphism, then e must be an isomorphism. If $A \xrightarrow{m} B$ is an extremal monomorphism, then (A, m) is called an extremal subobject of B.

The following is Proposition 17.11 in [6] and its dual.

Proposition B.24. *Every regular epimorphism is an extremal epimorphism, and every regular monomorphism is an extremal monomorphism.*

B.11. Factorizations. The following is a combination of Definitions 17.15 and 33.1 in [6].

Definition B.25. *Let E and M be classes of morphisms of a category $\mathcal{C}$.*

(1) A pair (e, m) is called an (E, M)-factorization of a $\mathcal{C}$-morphism f provided that $f = m \circ e$, where $e \in E$ and $m \in M$. We say that $f = m \circ e$ is an (E, M)-factorization of f.

(2) $\mathcal{C}$ is called an (E, M)-factorizable category if each $\mathcal{C}$-morphism has an (E, M)-factorization.

(3) $\mathcal{C}$ is called a uniquely (E, M)-factorizable category if and only if it is (E, M)-factorizable and for any two (E, M)-factorizations $f = m \circ e = \bar{m} \circ \bar{e}$, there exists an isomorphism h such that the diagram

commutes.

(4) $\mathcal{C}$ *is called an* (E, M) *category provided that it is uniquely* (E, M)*-factorizable and both* E *and* M *are closed under composition.*

Remark B.26. *If* E *is the class of extremal epimorphisms (resp. regular epimorphisms) of* $\mathcal{C}$ *and* M *is the class of all* $\mathcal{C}$*-monomorphisms, an* (E, M)*-factorization is called an (extremal epi, mono)-factorization (resp. (regular-epi,mono)-factorization), and if* E *is the class of all* $\mathcal{C}$*-epimorphisms and* M *is the class of all extremal monomorphisms (resp. regular monomorphisms) in* $\mathcal{C}$*, an* (E, M)*-factorization is called an (epi, extremal mono)-factorization (resp. (epi, regular mono)-factorization).*

The following is Proposition 17.18 in [6].

Proposition B.27. *If a category* $\mathcal{C}$ *is (regular epi, mono)-factorizable, then*

(1) $\mathcal{C}$ *is uniquely (regular epi, mono)-factorizable.*

(2) The regular epimorphisms in $\mathcal{C}$ *are precisely the extremal epimorphisms.*

The following is Proposition 33.4 in [6].

Proposition B.28. *For any category* $\mathcal{C}$*, the following are equivalent:*

(1) $\mathcal{C}$ *is (regular epi, mono)-factorizable.*

(2) $\mathcal{C}$ *is a (regular epi, mono) category.*

The following is a combination of Proposition 33.4 and Theorem 34.5 in [6].

Theorem B.29. *Every well-powered category* $\mathcal{C}$ *that has intersections and equalizers has the (epi, extremal mono)-diagonalization property and is an (epi, extremal mono) category.*

The following is Proposition 34.2 in [6].

Theorem B.30. *If* $\mathcal{C}$ *has the (epi, extremal mono)-diagonalization property, then in* $\mathcal{C}$:

(1) The composition of extremal monomorphisms is an extremal monomorphism.

(2) The intersection of extremal subobjects is an extremal subobject.

(3) The inverse image (pullback) of an extremal monomorphism is an extremal monomorphism.

(4) The product of extremal monomorphisms is an extremal monomorphism.

B.12. **Functors and Natural Transformations.** The following is a combination of Definitions 9.1 and 9.5 in [6].

Definition B.31. *Let* $\mathcal{C}$ *and* $\mathcal{D}$ *be categories. A (covariant) functor from* $\mathcal{C}$ *and* $\mathcal{D}$ *is a triple* $(\mathcal{C}, F, \mathcal{D})$ *where* F *is a function from the class of morphisms of* $\mathcal{C}$ *to the class of morphisms of* $\mathcal{D}$ *satisfying the following conditions:*

(1) F preserves identities; i.e., if 1_A is a $\mathcal{C}$-identity, then $F(1_A)$ is a $\mathcal{D}$-identity.

(2) F preserves composition; $F(f \circ g) = F(f) \circ F(g)$; i.e., whenever $dom(f) = cod(g)$, then $dom(F(f)) = cod(F(g))$ and the above equality holds.

We will often use $F : \mathcal{C} \to \mathcal{D}$ or $\mathcal{C} \xrightarrow{F} \mathcal{D}$ instead of $(\mathcal{C}, F, \mathcal{D})$ to denote a functor.

A triple $(\mathcal{C}, F, \mathcal{D})$ is called a contravariant functor from $\mathcal{C}$ to $\mathcal{D}$ if and only if $(\mathcal{C}^{op}, F, \mathcal{D})$ is a functor (or, equivalently, if and only if $(\mathcal{C}, F, \mathcal{D}^{op})$ is a functor).

The following is Definition 13.1 in [6].

Definition B.32. *Let* $F : \mathcal{A} \to \mathcal{B}$ *and* $G : \mathcal{A} \to \mathcal{B}$ *be functors.*

(1) A natural transformation (or functor morphism) from F to G is a triple (F, η, G) where $\eta : Ob(\mathcal{A}) \to Mor(\mathcal{B})$ is a function satisfying the following conditions:

(i) For each $\mathcal{A}$-object A, $\eta(A)$ (usually denoted by η_A) is a $\mathcal{B}$-morphism $\eta_A : F(A) \to G(A)$.

(ii) For each $\mathcal{A}$-morphism $A \xrightarrow{f} A'$, the diagram

$$F(A) \xrightarrow{\eta_A} G(A) \qquad A$$

$$\downarrow{F(f)} \qquad \downarrow{G(f)} \quad \downarrow{f}$$

$$F(A') \xrightarrow{\eta_{A'}} G(A') \qquad A'$$

commutes.

(2) A natural transformation (F, η, G) is called a natural isomorphism provided that for each $\mathcal{A}$-object A, η_A is a $\mathcal{B}$-isomorphism.

(3) F and G are said to be naturally isomorphic (denoted by $F \cong G$) if and only if there exists a natural isomorphism from F to G.

Notation B.33. *We will often write F in place of 1_F for the identity natural transformation on the functor F.*

B.13. Limits and Colimits. Both limits and colimits are important examples of universal objects. In this section, we recall the definitions of universals and limits and examine some examples.

As in [8] and Definition 26.1 in [6], the general notion of a universal arrow (map) is given by the following definition.

Definition B.34. *If $S \colon \mathcal{D} \longrightarrow \mathcal{C}$ is a functor and c an object of $\mathcal{C}$, a universal arrow from c to S (a universal map for c with respect to S) is a pair (r, u) consisting of an object r of $\mathcal{D}$ and an arrow $u \colon c \longrightarrow S(r)$ of $\mathcal{C}$, such that to every pair (d, f) with d an object of $\mathcal{D}$ and $f \colon c \longrightarrow S(d)$ an arrow of $\mathcal{C}$, there is a unique arrow $f' \colon r \longrightarrow d$ of $\mathcal{D}$ with $S(f') \circ u = f$. In other words, every arrow f to S factors uniquely through the universal arrow u, as in the following commutative diagram:*

$$c \xrightarrow{u} S(r) \qquad r$$

$$\downarrow{1_c} \qquad \downarrow{S(f')} \quad \downarrow{f'}$$

$$c \xrightarrow{f} S(d) \qquad d$$

Remark B.35. *By Proposition 26.7 in [4], the object r is unique up to isomorphism in $\mathcal{D}$.*

The following is the dual of the above definition.

Definition B.36. *If $S\colon \mathcal{D} \longrightarrow \mathcal{C}$ is a functor and c an object of $\mathcal{C}$, a universal arrow from S to c (a co-universal map for c with respect to S) is a pair (r, v) consisting of an object $r \in \mathcal{D}$ and an arrow $v\colon S(r) \longrightarrow c$ of $\mathcal{C}$, such that to every pair (d, f) with $d \in \mathcal{D}$ and $f\colon S(d) \longrightarrow c$ an arrow of $\mathcal{C}$, there is a unique arrow $f'\colon d \longrightarrow r$ of $\mathcal{D}$ with $v \circ S(f') = f$. This gives the commutative diagram*

$$
\begin{array}{ccc}
d & S(d) \xrightarrow{\ f\ } & c \\
\downarrow{f'} & \downarrow{S(f')} & \downarrow{1_c} \\
r & S(r) \xrightarrow{\ v\ } & c
\end{array}
$$

This comes from page 67 in [8].

Definition B.37. *Let C and J be categories, and let C^J denote the category of functors between J and C. For each $c \in C$, we let $\Delta c\colon J \longrightarrow C$ denote the functor defined via $\Delta c(i) = c$ for each $i \in J$ and $\Delta c(f) = 1_c$ for each arrow f in J. Furthermore, for an arrow $f\colon c \longrightarrow c'$ of C, we define $\Delta f\colon \Delta c \longrightarrow \Delta c'$ to be the natural transformation which has the same value f at each object $i \in J$. It is clear that Δf is indeed is a natural transformation since the diagram*

$$
\begin{array}{ccc}
\Delta c(i) = c & \xrightarrow{\Delta f_i = f} & \Delta c'(i) = c' \\
\downarrow{1_c} & & \downarrow{1_c'} \\
\Delta c(j) = c & \xrightarrow{\Delta f_j = f} & \Delta c'(j) = c'
\end{array}
$$

commutes for each $i, j \in J$. We define the diagonal functor $\Delta\colon C \longrightarrow C^J$ via $c \mapsto \Delta c$ and $f \mapsto \Delta f$.

Definition B.38. *A universal arrow (r, ν) from Δ to a functor $F \in C^J$ is called a limit for the functor F. It consists of an object $r \in C$ and a natural transformation $\nu\colon \Delta r \longrightarrow F$ which is universal among all natural transformations $\tau\colon \Delta c \to F$. A universal arrow (r, μ) from a functor $F \in C^J$ to Δ is called a colimit for the functor F. It consists of an object $r \in C$ and a natural transformation $\mu\colon F \longrightarrow \Delta r$ which is universal among all natural transformations $\tau\colon F \longrightarrow \Delta c$.*

Remark B.39. *Since Δc is the constant functor, each natural transformation $\tau : \Delta c \longrightarrow F$ consists of one arrow $\tau_i \colon c \to F_i$ for each $i \in J$ making the diagrams*

$$
\begin{array}{ccc}
c & \xrightarrow{\tau_i} & F_i \\
\downarrow{\scriptstyle 1_c} & & \downarrow{\scriptstyle F_u} \\
c & \xrightarrow{\tau_j} & F_j
\end{array}
$$

commute for each $i, j \in J$, where $u \colon i \to j$ is any arrow in J. It is useful to depict these diagrams with the c's identified as follows:

$$
\begin{array}{ccc}
& & F_i \\
& {\scriptstyle \tau_i}\nearrow & \downarrow{\scriptstyle F_u} \\
c & \xrightarrow[\tau_j]{} & F_j
\end{array}
$$

Because of their visual appearance, a natural transformation $\tau : \Delta c \to F$ is called a cone to the base F from the vertex c.

Hence, a limit of $F : J \to C$ consists of an object $Lim(F) \in C$ and a cone $\nu : \Delta Lim(F) \to F$ to the base F from the vertex $Lim(F)$ which is universal; that is, for any cone $\tau : \Delta c \to F$ to the base F from the vertex c, there is a unique arrow $t : c \to Lim(F)$ with $\tau_i = \nu_i t$ for every $i \in J$. We say ν is the limiting cone or universal cone to F.

Remark B.40. *Since Δc is the constant functor, each natural transformation $\tau \colon F \longrightarrow \Delta c$ consists of arrows $\tau_i \colon F_i \longrightarrow c$ making the diagrams*

$$
\begin{array}{ccc}
F_i & \xrightarrow{\tau_i} & c \\
\downarrow{\scriptstyle F_u} & & \downarrow{\scriptstyle 1_c} \\
F_j & \xrightarrow{\tau_j} & c
\end{array}
$$

commute for each $i, j \in J$, where $u \colon i \to j$ is any arrow in J. It is useful to view these diagrams with the c's identified as follows:

$$
\begin{array}{ccc}
F_i & & \\
\Big\downarrow{\scriptstyle F_u} & \searrow{\scriptstyle \tau_i} & \\
F_j & \xrightarrow{\;\;\tau_j\;\;} & c
\end{array}
$$

Because of their visual appearance, a natural transformation $\tau\colon F \to \Delta c$ is called a cone from the base F to the vertex c.

Hence, a colimit of $F : J \to C$ consists of an object $Colim(F) \in C$ and a cone $\mu : F \to \Delta Colim(F)$ from the base F to the vertex $Colim(F)$ which is universal; that is, for any cone $\tau : F \to \Delta c$ from the base F to the vertex c, there is a unique arrow $t : Colim(F) \to c$ with $\tau_i = t\mu_i$ for every $i \in J$. We say μ is the limiting cone or universal cone from F.

The following comes from [4].

Definition B.41. *A diagram D in a category $\mathcal{C}$ is a directed graph whose vertices $i \in I$ are labeled by objects R_i in $\mathcal{C}$ and whose edges $i \longrightarrow j$ are labeled by morphisms in $Hom_{\mathcal{C}}(R_i, R_j)$. The underlying graph is called the scheme of the diagram.*

As in [4], we can define cones and limits in terms of diagrams instead of functors.

Definition B.42. *A family of morphisms $(f_i : A \to R_i)_{i \in I}$ with common domain A is a cone for D, provided that for each arrow $d : R_i \to R_j$ in D, the diagram*

$$
\begin{array}{ccc}
 & & R_i \\
 & \nearrow{\scriptstyle f_i} & \Big\downarrow{\scriptstyle d} \\
A & \xrightarrow[\;f_j\;]{} & R_j
\end{array}
$$

commutes. A limit for D is a cone for D with the universal property that any other cone for D uniquely factors through it.

B.13.1. *Products and Coproducts.* Recall from Definition 3.4 in [6] that a category is said to be complete if all of its morphisms are identities. The following comes from page 69 in [8].

Definition B.43. *If J is a discrete category, then a functor $F : J \to C$ is a J-indexed family of objects a_j in C, while a cone with vertex c and base a_j is just a J-indexed family of arrows $f_j : c \longrightarrow a_j$. A universal cone $p_j : \prod_j a_j \longrightarrow a_j$ thus consists of an object $\prod_j a_j$, called the product of the factors a_j and of arrows p_j, called the projections of the product, with the following universal property: To each J-indexed family (i.e. cone) $f_j : c \longrightarrow a_j$ there is a unique $f : c \longrightarrow \prod_j a_j$ with $p_j f = f_j$ for each $j \in J$.*

B.13.2. *Equalizers and Coequalizers.* The following comes from page 70 in [8].

Definition B.44. *If J is the category with precisely two objects and two non-identity arrows from the first object to the second, then a functor $F : J \to C$ is a pair $f, g : a \to b$ of parallel arrows in C. A limit object of F is called an equalizer (or difference kernel) of f and g.*

Note that any equalizer e is monic by the uniqueness of a limit.

B.13.3. *Pullbacks and Pushouts.* The following comes from page 71 in [8].

Definition B.45. *Let J be the category with precisely three objects and having precisely two non-identity arrows, with one from the first object to the second, and the other from the third to the second. Then a functor $F : J \to C$ is a pair of arrows in C with a common codomain a. A cone over F is a pair of arrows h, k from some $c \in C$ making the diagram*

$$
\begin{array}{ccc}
c & \xrightarrow{k} & d \\
\downarrow h & & \downarrow g \\
b & \xrightarrow{f} & a
\end{array}
$$

commute. A universal cone is a commutative square of this form and is called a pullback square, and the vertex c of the square is called a pullback. The pullback of a pair of equal arrows is called the kernel pair of f.

As in [6], h is said to be a pullback of g along f. If g is a monomorphism, h is commonly called an inverse image of g along f. The dual notion of a pullback is a pushout.

Definition B.46. *A category is called regular if it has finite limits, coequalizers, and if the pullback of a regular epimorphism is always a regular epimorphism.*

The following is Theorem 21 in [4].

Theorem B.47. *The categories $\mathfrak{P}$ and $\mathfrak{C}$ are regular.*

B.14. **Completeness and Cocompleteness.** The following is Definition 23.1 in [6].

Definition B.48. *Let $\mathfrak{C}$ be a category.*

(1) If I is a category, then $\mathfrak{C}$ is said to be I-complete (or to have I-limits) provided that every functor $D : I \longrightarrow \mathfrak{C}$ has a limit.

(2) $\mathfrak{C}$ is said to be complete provided that $\mathfrak{C}$ is I-complete for each small category I.

(3) $\mathfrak{C}$ is said to be finitely complete (or to have finite limits) provided that $\mathfrak{C}$ is I-complete for each finite category I.

The corresponding dual notions are I^{op}-cocomplete, cocomplete, and finitely cocomplete.

The following is Theorem 23.8 in [6].

Theorem B.49. *For any category, $\mathfrak{C}$, the following are equivalent:*

(1) $\mathfrak{C}$ is complete.

(2) $\mathfrak{C}$ has multiple pullbacks and terminal objects.

(3) $\mathfrak{C}$ has products and pullbacks.

(4) $\mathcal{C}$ has products and inverse images.

(5) $\mathcal{C}$ has products and finite intersections.

(6) $\mathcal{C}$ has products and equalizers.

(7) $\mathcal{C}$ has products, equalizers, and intersections of regular subobjects.

(8) $\mathcal{C}$ is finitely complete and has inverse limits.

B.15. Projective and Injective Objects. The following comes from Theorems 12.15 and 12.16 in [6].

Theorem B.50. *An object P is $\mathcal{C}$-projective provided that for each $\mathcal{C}$-epimorphism $\mathcal{B}\xrightarrow{f}C$ and each morphism $P\xrightarrow{g}C$, there exists a morphism $P\xrightarrow{h}B$ making the triangle*

$$
\begin{array}{ccc}
 & & P \\
 & \overset{h}{\swarrow} & \downarrow g \\
B & \underset{f}{\longrightarrow} & C
\end{array}
$$

commute. By duality, an object Q is $\mathcal{C}$-injective provided that for each $\mathcal{C}$-monomorphism $C\xrightarrow{f}B$ and each morphism $C\xrightarrow{g}Q$, there exists a morphism $\mathcal{B}\xrightarrow{h}Q$ such that the triangle

$$
\begin{array}{ccc}
C & \overset{f}{\longrightarrow} & B \\
g \downarrow & \swarrow h & \\
Q & &
\end{array}
$$

commutes.

B.16. Separators and Coseparators. The following comes from Theorems 12.15 and 12.16 in [6].

Theorem B.51. *A $\mathcal{C}$-object S is a $\mathcal{C}$-separator if and only if whenever $A\xrightarrow{f}B$ and $A\xrightarrow{g}B$ are distinct $\mathcal{C}$-morphisms, there exists a $\mathcal{C}$-morphism $S\xrightarrow{x}A$ such that $S\xrightarrow{x}A\xrightarrow{f}B \neq S\xrightarrow{x}A\xrightarrow{g}B$.*

By duality, a $\mathcal{C}$-object C is a $\mathcal{C}$-coseparator if and only if whenever $A\xrightarrow{f}B$ and $A\xrightarrow{g}B$ are distinct $\mathcal{C}$-morphisms, there exists a $\mathcal{C}$-morphism $\mathcal{B}\xrightarrow{x}C$ such that $A\xrightarrow{f}B\xrightarrow{x}C \neq A\xrightarrow{g}B\xrightarrow{x}C$.

APPENDIX C

MAPLE CODE FOR PLOTTING PRECODES

Let A be a precode. This section gives Maple code for representing and plotting A. It also gives code for constructing A^{op}, $A_\#$, $A_{\mathsf{V}\mathsf{I}}$, and A_{II}.

We plot precodes using the plots and plottools packages. Thus, we need to include these packages using the following commands.

```
>  with(plots):
>  with(plottools):
```

We represent precodes using a custom data structure. We create this structure using the following command.

```
>  new_precode(A):
```

We add plaintext and codetext elements to the precode using the add_plain() and add_code() commands, respectively. For example, the commands

```
>  add_plain(A,{p1,p2}):
>  add_code(A,{c1,c2,c3}):
```

add the elements $p1$ and $p2$ to P_A and add $c1$, $c2$, and $c3$ to C_A. We add edges to the precode using the add_edge() command. For example, the command

```
>  add_edge(A,[[p1,c2,e]])
```

adds to the precode A a directed edge from $p1$ to $c2$; that is, it adds $(p1, c2)$ to e_A. The command

```
>  add_edge(A,[[p1,c2,d]])
```

adds to the precode A a directed edge from $c2$ to $p1$; that is, it adds $(c2, p1)$ to d_A. The command

```
>  add_edge(A,[[p1,c2,s]])
```

adds both edges to A. It is important to note that the command

```
>  update_vdata(A)
```

must be executed after the vertices and edges have been defined to update the rest of the data in the precode data structure. We can use the $eval(A)$ command to display the structure of A.

For example, if we construct A using the commmands

```
>  new_precode(A):
>  add_plain(A,{p1,p2}):
>  add_code(A,{c1,c2,c3}):
>  add_edge(A,[[p1,c2,e],[p1,c3,d],[p2,c2,s],[p1,c1,e],[p1,c1,d]]):
>  update_vdata(A):
```

then the command

```
>  eval(A)
```

produces the following output.

$TABLE([\_PData = TABLE([p1 = [1, 1, 1], p2 = [0, 0, 1]]),$
$\_CList = vector([TABLE([]), TABLE([]), TABLE([1 = c3]), TABLE([]), TABLE([1 = c1]),$
$TABLE([1 = c2]), TABLE([]), TABLE([])]),$
$\_PNum = vector([0, 0, 0, 0, 1, 0, 0, 1]), \_CNbrs = TABLE([c1 = p1, c2 = p1, p2, c3 = p1]),$
$\_CNum = vector([0, 0, 1, 0, 1, 1, 0, 0]),$
$\_EData = TABLE([(p1, c2) = [1, 0], (p1, c1) = [1, 1], (p1, c3) = [0, 1], (p2, c2) = [1, 1]]),$
$\_C = c1, c2, c3, \_P = p1, p2,$
$\_CData = TABLE([c1 = [0, 0, 1], c2 = [1, 0, 1], c3 = [0, 1, 0]]),$
$\_E = [p1, c2], [p1, c3], [p2, c2], [p1, c1], \_PNbrs = TABLE([p1 = c1, c2, c3, p2 = c2]),$

$$\_PList = vector([TABLE([]), TABLE([]), TABLE([]), TABLE([]),$$
$$TABLE([1 = p2]), TABLE([]), TABLE([]), TABLE([1 = p1])])]);$$

Algorithm C.1. *In this procedure, we create a precode data structure A. A is a table with entries defined as follows:*

$\_P$ is the set of plaintext elements.

$\_C$ is the set of codetext elements.

$\_PData$ is a table indexed by the elements in $\_P$. It is initialized by executing *update_vdata(A)*.

  For a plaintext element p, $A[\_PData][op(p)]$ is a three-member list such that

    $A[\_PData][op(p)][1] = 1$ if p is part of an encode edge which is NOT in the converse of the decode relation. It is 0 otherwise.

    $A[\_PData][op(p)][2] = 1$ if p is part of a decode edge which is NOT in the converse of the encode relation. It is 0 otherwise.

    $A[\_PData][op(p)][3] = 1$ if p is part of an encode edge which is also in the converse of the decode relation. It is 0 otherwise.

$\_CData$ is a table indexed by the elements in $\_C$. It is initialized by executing *update_vdata(A)*.

  For a codetext element c, $A[\_CData][op(c)]$ is a three-member list such that

    $A[\_CData][op(c)][1] = 1$ if c is part of an encode edge which is NOT in the converse of the decode relation. It is 0 otherwise.

    $A[\_CData][op(c)][2] = 1$ if c is part of a decode edge which is NOT in the converse of the encode relation. It is 0 otherwise.

    $A[\_CData][op(c)][3] = 1$ if c is part of an encode edge which is also in the converse of the decode relation. It is 0 otherwise.

$\_PNum$ is an eight element array initialized by *update_vdata(A)*.

  $A[\_PNum][P3 * (4) + P2 * (2) + P1 + 1]$ contains the number of p for which

    $A[\_PData][op(p)] = [P1, P2, P3]$. In particular,

  $A[\_PNum][8]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [1, 1, 1]$.

  $A[\_PNum][7]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [1, 1, 0]$.

  $A[\_PNum][6]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [1, 0, 1]$.

  $A[\_PNum][5]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [1, 0, 0]$.

  $A[\_PNum][4]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [0, 1, 1]$.

  $A[\_PNum][3]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [0, 1, 0]$.

  $A[\_PNum][2]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [0, 0, 1]$.

  $A[\_PNum][1]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [0, 0, 0]$.

$\_CNum$ is an eight element array initialized by *update_vdata(A)* defined analogously to $\_PNum$ to enumerate the codetext elements of each type.

$\_PList$ is an eight element array initialized by *update_vdata(A) such* that

  $A[\_PList][P3 * (4) + P2 * (2) + P1 + 1]$ contains the number of p for which

    $A[\_PData][op(p)] = [P1, P2, P3]$. In particular,

  $A[\_PList][8]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [1,1,1]$.

  $A[\_PList][7]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [1, 1, 0]$.

  $A[\_PList][6]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [1, 0, 1]$.

  $A[\_PList][5]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [1, 0, 0]$.

  $A[\_PList][4]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [0, 1, 1]$.

  $A[\_PList][3]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [0, 1, 0]$.

  $A[\_PList][2]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [0, 0, 1]$.

  $A[\_PList][1]$ contains the number of plaintext elements p for which $A[\_PData][op(p)] = [0, 0, 0]$.

$\_CList$ is an eight element array initialized by update_vdata(A) defined analogously to $\_PList$ to list the codetext elements of each type.

$\_PNbrs$ is a table indexed by the elements in $\_P$. It is updated by each call to *add_edge()*.

  For a plaintext element p, $A[\_PNbrs][op(p)]$ is the set of the codetext elements adjacent to p.

CNbrs is a table indexed by the elements in _C_. It is updated by each call to _add_edge()_.

For a plaintext element c, $A[\_CNbrs][op(c)]$ is the set of the plaintext elements adjacent to c. _E_ is the set of edges.

Each edge is of the form $[p, c]$, where p is a plaintext element and c is a codetext element. _EData_ is a table indexed by the elements in _E_. It is initialized by each call to _add_edge(A)_.

For an edge $[p, c]$, $A[\_EData][p, c]$ is a two-member list such that

$A[\_EData][p, c][1] = 1$ if (p, c) is in the encode relation of A. It is 0 otherwise.

$A[\_EData][p, c][1] = 2$ if (c, p) is in the decode relation of A. It is 0 otherwise.

```
new_precode:=proc(A) local i;
    A:=table():
    A[_P]:={}:
    A[_C]:={}:
    A[_E]:={};
    A[_PData]:=table():
    A[_CData]:=table():
    A[_EData]:=table():
    A[_PNbrs] := table():
    A[_CNbrs] := table():
    A[_PNum] := array(1..8);
    A[_CNum] := array(1..8);
    A[_PList] := array(1..8);
    A[_CList] := array(1..8);
    eval(A);
end proc:
```

Algorithm C.2. *This procedure adds the plaintext elements in the set p_set to the precode A.*

```
add_plain:=proc(A,p_set::set) local i;
    #Add the plaintext elements to _P.
    A[_P] := {op(A[_P]),op(p_set)};
    #Add the corresponding entries to the table _PNbrs.
    for i from 1 to nops(p_set) do
        A[_PNbrs][p_set[i]]:={};
    end do;
    eval(A);
end proc:
```

Algorithm C.3. *This procedure adds codetext elements in the set c_set to the precode A.*

```
add_code:=proc(A,c_set::set) local i;
    #Add the codetext elements to _C.
    A[_C] := {op(A[_C]),op(c_set)};
    #Add the corresponding entries to the table _CNbrs.
    for i from 1 to nops(c_set) do
        A[_CNbrs][c_set[i]]:={};
    end do;
    eval(A);
end proc:
```

Algorithm C.4. *This procedure adds the edges in the list e_list to the precode A. An edge in e_list has the form $[p, c, type]$, where p is a plaintext element in A, c is a codetext element in A, and type is either s, e, or d.*

If type = s, then (p, c) is in the encode relation of A, and (c, p) is in the decode relation of A. If type = e, then (p, c) is in the encode relation of A. If type = d, then (c, p) is in the decode relation of A.

The following variables are used in this procedure:

Num is the number of entries in *e_list*.
edge is an entry in *e_list*.
p is the plaintext element in *edge*.
c is the codetext element in *edge*.
edge_type is the *edge* type.
i is a loop index.

```
add_edge:=proc(A,e_list::listlist) local Num, edge, p, c, edge_type,i;
    Num := nops(e_list);
    for i from 1 to Num do
        edge := e_list[i];
        p := edge[1];
        c := edge[2];
        edge_type := edge[3];
        # Make sure the edge contains valid endpoints.
        if not member(p,A[_P]) or not member(c,A[_C]) then
            print("Error: edge", edge, "contains invalid vertices.");
            print("Plaintext:",A[_P],"Codetext:",A[_C]);
            return;
        end if;

        # Make sure the set of neighbors is correct.
        A[_PNbrs][p]:={op(A[_PNbrs][p]),c};
        A[_CNbrs][c]:={op(A[_CNbrs][c]),p};

        #Add the edge entry if it is not present.
        if not member([p,c],A[_E]) then
            A[_E] := {op(A[_E]), [p,c]};
            A[_EData][p,c]:=[0,0];
        end if;

        #Update the edge entry.
        if edge_type = e then
            A[_EData][p,c][1]:=1;
        elif edge_type = d then
            A[_EData][p,c][2]:=1;
        else
            A[_EData][p,c]:=[1,1];
        end if;
    end do;
    eval(A);
end proc:
```

Algorithm C.5. *This procedure updates the plaintext and codetext data for the precode A. It should be executed once all the edges have been added to A.*

The following variables are used in this procedure:

Num is the number of entries in e_list.
p is a plaintext element.
c is a codetext element.
$edge$ is an edge in A.
$edge\_type$ contains the $edge$ type.
The list $[P1, P2, P3]$ contains $A[\_PData][p]$.
The list $[C1, C2, C3]$ contains $A[\_CData][c]$.
$index$ contains $C3 * (4) + C2 * (2) + C1 + 1$ or $P3 * (4) + P2 * (2) + P1 + 1$.
i is a loop index.

```
update_vdata:=proc(A) local edge_type,edge,p,c,P1,P2,P3,C1,C2,C3,index,i;
    #Initialize the data.
    for i from 1 to 8 do
        A[_PList][i]:=table();
        A[_CList][i]:=table();
        A[_PNum][i]:=0;
        A[_CNum][i]:=0;
    end do;
    for i from 1 to nops(A[_P]) do
        A[_PData][A[_P][i]]:=[0,0,0];
    end do;
    for i from 1 to nops(A[_C]) do
        A[_CData][A[_C][i]]:=[0,0,0];
    end do;

    #Update the vertex data.
    for i from 1 to nops(A[_E]) do
        edge := A[_E][i];
        p := edge[1];
        c := edge[2];
        edge_type := A[_EData][p,c];
        if edge_type = [1,1] then
            # There is a non-directed edge (type s) incident on p and c.
            A[_PData][p][3] := 1;
            A[_CData][c][3] := 1;
        elif edge_type = [1,0] then
            # There is an edge of type e incident on p and c.
            A[_PData][p][1] := 1;
            A[_CData][c][1] := 1;
        else
            # There is an edge of type d incident on p and c.
            A[_PData][p][2] := 1;
            A[_CData][c][2] := 1;
        end if;
    end do;

    # Classify each codetext element based on the type of edges incident on it.
    # That is, update A[_CNum] and A[_CList].
    for i from 1 to nops(A[_C]) do
```

```
        c := A[_C][i];
        C1 := A[_CData][c][1];
        C2 := A[_CData][c][2];
        C3 := A[_CData][c][3];
        # Find the value (+1) of the binary number represented by C3 C2 C1.
        index := C3*(4)+C2*(2)+C1+1;
        A[_CNum][index] := A[_CNum][index] + 1;
        A[_CList][index][A[_CNum][index]]:=c;
    end do;

    # Classify each plaintext element based on the type of edges incident on it.
    # That is, update A[_PNum] and A[_PList].
    for i from 1 to nops(A[_P]) do
        p := A[_P][i];
        P1 := A[_PData][p][1];
        P2 := A[_PData][p][2];
        P3 := A[_PData][p][3];
        # Find the value (+1) of the binary number represented by P3 P2 P1.
        index := P3*(4)+P2*(2)+P1+1;
        A[_PNum][index] := A[_PNum][index] + 1;
        A[_PList][index][A[_PNum][index]]:=p;
    end do;
    eval(A);
end proc:
```

Algorithm C.6. *This procedure returns a copy of the precode A.*

The following variables are used in this procedure:
H will contain the copy of A.
edge is an edge in *A*.
edge_type contains the *edge* type.
i is a loop index.

```
copy_precode:=proc(A) local H,type,edge,i;

    new_precode(H): #Create the precode.
    add_plain(H,A[_P]); #Add the plaintext elements to H.
    add_code(H,A[_C]); #Add the codetext elements to H.

    #Add the edges to H.
    for i from 1 to nops(A[_E]) do
        edge := A[_E][i];
        type := A[_EData][op(edge)];
        if type = [1,1] then
            #The edge is of type s.
            add_edge(H,[[edge[1],edge[2],s]]);
        elif type = [1,0] then
            #The edge is of type e.
            add_edge(H,[[edge[1],edge[2],e]]);
        else
            #The edge is of type d.
            add_edge(H,[[edge[1],edge[2],d]]);
        end if;
```

```
        end do;

        #Update the vertex data.
        update_vdata(H);
        return H;
    end proc:
```

C.1. The Plotting Algorithms. The following procedures are used to plot a precode.

Algorithm C.7. *This procedure orders the vertices in A to produce a more eye-pleasing plot.*

The following variables are used in this procedure:

P_List will contain the ordered list of plaintext elements.
C_List will contain the ordered list of codetext elements.
P contains the list of plaintext elements yet to be handled.
c is a codetext element.
$Nbrs$ is the list of plaintext elements adjacent to c.
$NumNbrs$ is the number of elements in $Nbrs$.
The list $[P1, P2, P3]$ contains $A[\_PData][Nbrs[k]]$ (k runs from 1 to $NumNbrs$).
$index$ contains $P3 * (4) + P2 * (2) + P1 + 1$.
$PList$ is an array of 8 elements such that $PList[index]$ contains
    the elements in P which are incident on c.
i, j, and k are loop indices.

```
plot_order:=proc(A) local c,P,PList,C_List,P_List,Nbrs,NumNbrs,P1,P2,P3,i,j,k,index;

    PList := array(1..8);
    C_List := [];
    P_List := [];
    P:=A[_P];

    # j represents the type of codetext element to be handled.
    # Recall the definitions of A[_CNum] and A[_CList].
    for j from 8 to 1 by -1 do
        # Loop on each codetext element of the current type.
        for i from 1 to A[_CNum][j] do
            c := A[_CList][j][i];
            C_List := [c,op(C_List)];
            for k from 1 to 8 do
                PList[k] := [];
            end do;
            Nbrs := A[_CNbrs][c];
            NumNbrs := nops(Nbrs);

            # Loop on the plaintext elements adjacent to c.
            for k from 1 to NumNbrs do
                # Make sure Nbrs[k] hasn't been handled already.
                if member(Nbrs[k],P,'m') then
                    P:=subsop(m=NULL,P);
                    P1 := A[_PData][Nbrs[k]][1];
                    P2 := A[_PData][Nbrs[k]][2];
                    P3 := A[_PData][Nbrs[k]][3];
```

```
                # Find the value (+1) of the binary number represented by P3 P2 P1.
                # Add Nbrs[k] to the appropriate list based on its type.
                index := P3*(4)+P2*(2)+P1+1;
                PList[index] := [Nbrs[k],op(PList[index])];
            end if;
        end do;
        # Form P_List from the 8 lists in PList.
        for k from 8 to 1 by -1 do
            P_List := [op(PList[k]),op(P_List)];
        end do;
    end do;
end do;

    # Add the isolated plaintext elements to the list.
    P_List := [op(P),op(P_List)];
    return P_List,C_List;
end proc:
```

Algorithm C.8. *This procedure plots the graph of the precode A based on the order specified by plot_order(A). The parameter title_string gives the string with which the plot will be labeled and type specifies whether the plot should be a STRIP chart or a BIPARTITE_GRAPH (see below).*

The following are the variables used in this procedure:

STRIP is a constant with value 0 representing a strip chart display.
BIPARTITE_GRAPH is a constant with value 1 representing a bipartite graph display.

S contains *plot_order(A)*.
    P contains $S[1]$, the list of plaintext elements in order.
    C contains $S[2]$, the list of codetext elements in order.

PL is the set of plaintext elements. It has length *NumPL*.
CO is the set of codetext elements. It has length *NumCO*.
E contains the edge list. It has length *NumE*.

Height is the height of the graph.
Width is the width of the graph.
x1 contains the x-coordinate of the left plaintext column
    (the only plaintext column if the plot is a *BIPARTITE_GRAPH*).
x2 contains the x-coordinate of the codetext column.
x3 contains the x-coordinate of the right plaintext column in a *STRIP* chart plot.
yp_step contains the vertical distance between each plaintext vertex.
yc_step contains the vertical distance between each codetext vertex.
yp contains the y-coordinate of the current plaintext vertex.
yc contains the y-coordinate of the current codetext vertex.

PPoints is a table indexed by the plaintext elements which contains the y-coordinate of
    each vertex.
CPoints is a table indexed by the codetext elements which contains the y-coordinate of
    each vertex.
Points_Structures will contain the POINTS plot structures necessary to display the vertices.
Points_List is a list of coordinates used to create the POINTS plot structures.

Curves_Structures will contain the CURVES plot structures necessary to display the edges.
Text_Structures will contain the TEXT plot structures necessary to display the labels on
   the vertices.
   plain_label_offset gives the x-offset of the label for the plaintext elements in the first column.
VPlot will contain the plot of the precode.
edge contains the current edge.
i and *j* are loop indices.

precode_plot:=proc(A,title_string,type) local S,P,C,PL,NumPL,CO,NumCO,E,NumE,Height,Width,
x1,x2,x3,yp,yc,yp_step,yc_step,PPoints,CPoints,Points_List,Text_Structures,Points_Structures,
Curves_Structures,VPlot,edge,i,j,STRIP,BIPARTITE_GRAPH,plain_label_offset;

```
    STRIP := 0;
    BIPARTITE_GRAPH := 1;

    # Initialize the variables.
    S:=plot_order(A);
    P:=S[1];
    C:=S[2];

    PL:=A[_P];
    NumPL := nops(PL);
    CO:=A[_C];
    NumCO := nops(CO);
    E:=A[_E];
    NumE := nops(E);

    PPoints := table();
    CPoints := table();
    Text_Structures := [];
    Points_Structures := [];
    VPlot := [];

    # We now initialize the coordinates of the plot.
    Height:=max(NumCO,NumPL);

    # x1 contains the x coordinate of the left plaintext column.
    # x2 contains the x coordinate of the codetext column.
    # x3 contains the x coordinate of the right plaintext column in a STRIP chart plot.
    x1:=0.0;
    x2:=Height/2.0;
    x3:=Height;
    if type = STRIP then
        Width := x3;
    else
        Width := x2;
    end if;
    plain_label_offset := 0.05*Width;

    # We now initialize yp_step and yc_step.
    if NumPL = Height then
```

```
# The number of plaintext vertices is greater than or equal to the number of
# codetext elements.
yp_step := 1;
if NumCO = Height then
    yc_step:=1;
else
    yc_step:=(Height-1)/(NumCO+1);
end if;
else
    # The number of plaintext vertices is less than the number of codetext elements.
    yc_step:=1;
    yp_step:=(Height-1)/(NumPL+1);
end if;

# We now initialize yp and yc.
if NumCO = Height then
    # The number of codetext vertices is greater than or equal to the number of
    # plaintext elements.
    yc:=0;
    # Since the plaintext and codetext elements are not usually the same, it is aesthetically
    # pleasing to ensure that the plaintext and codetext elements are not aligned horizontally.
    if NumPL = Height then
        yp:=0.5;
    else
        yp:=yp_step;
    end if;
else
    # The number of codetext vertices is less than the number of plaintext elements.
    yc:=yc_step;
    yp:=0;
end if;

# The following segment initializes CPoints and the POINTS and TEXT plot structures
# necessary to plot the codetext elements.
Points_List := [];
for i from 1 to nops(C) do
    CPoints[C[i]]:=yc;
    Points_List := [op(Points_List),[x2,yc]];
    Text_Structures:=[op(Text_Structures),TEXT([x2,yc],convert(C[i],string),
        ALIGNBELOW,ALIGNRIGHT,FONT(TIMES,ROMAN,10))];
    yc:=yc+yc_step;
end do;
Points_Structures := [op(Points_Structures),POINTS(op(Points_List),SYMBOL(DIAMOND))];

# The following segment initializes PPoints and the POINTS and TEXT plot structures
# necessary to plot the plaintext elements.
Points_List := [];
for i from 1 to nops(P) do
    PPoints[P[i]]:=yp;
    Points_List := [op(Points_List),[x1,yp]];
    Text_Structures := [op(Text_Structures),
```

```
            TEXT([x1-plain_label_offset,yp],convert(P[i],string), ALIGNBELOW,
            ALIGNRIGHT,FONT(TIMES,ROMAN,10))];
     if type = STRIP then
         Points_List := [op(Points_List),[x3,yp]];
         Text_Structures := [op(Text_Structures),
             TEXT([x3,yp],convert(P[i],string), ALIGNBELOW,
             ALIGNRIGHT,FONT(TIMES,ROMAN,10))];
     end if;
     yp:=yp+yp_step;
end do;
Points_Structures :=[op(Points_Structures),POINTS(op(Points_List),SYMBOL(CIRCLE))];

# Build the CURVES plot structure necessary to plot the edges.
Curves_Structures := [];
for i from 1 to NumE do
    edge := E[i];
    yp := PPoints[edge[1]];
    yc := CPoints[edge[2]];
    if A[_EData][op(edge)] = [1,1] then
        # edge is undirected.
        if type = STRIP then
            Curves_Structures := [op(Curves_Structures),CURVES([[x1,yp], [x2,yc], [x3,yp]],
                THICKNESS(2),LINESTYLE(1), COLOR(RGB,1.0,0.0,0.0))];
        else
            Curves_Structures := [op(Curves_Structures),
                CURVES([[x1,yp], [x2,yc]], THICKNESS(2),LINESTYLE(1))];
        end if;
    elif A[_EData][op(edge)] = [1,0] then
        # edge is an encode edge
        if type = STRIP then
            Curves_Structures := [op(Curves_Structures),
                CURVES([[x1,yp],[x2,yc]], THICKNESS(2), LINESTYLE(1))];
        else
            Curves_Structures := [op(Curves_Structures),
                arrow([x1,yp],[x2,yc],.01, .1,.05,color=cyan)];
        end if;
    else
        # edge is a decode edge
        if type = STRIP then
            Curves_Structures := [op(Curves_Structures),
                CURVES([[x2,yc],[x3,yp]], THICKNESS(2), LINESTYLE(1))];
        else
            Curves_Structures := [op(Curves_Structures),
                arrow([x2,yc],[x1,yp],.01, .1,.05,color=cyan)];
        end if;
    end if;
end do;

# Put the pieces together to form the plot.
VPlot := [op(VPlot),PLOT(op(Points_Structures),op(Text_Structures),
    TEXT([Width/2.0,Height],title_string,ALIGNBELOW,
```

FONT(TIMES,ROMAN,14)),op(Curves_Structures),AXESSTYLE(NONE))];
display(VPlot,view=[-plain_label_offset..Width,0..Height]);
end proc:

In our example above, the commands
```
>  precode_plot(A,"Strip Chart",0);
>  precode_plot(A,"Bipartite Graph",1);
```
produce the output in Figure 30.
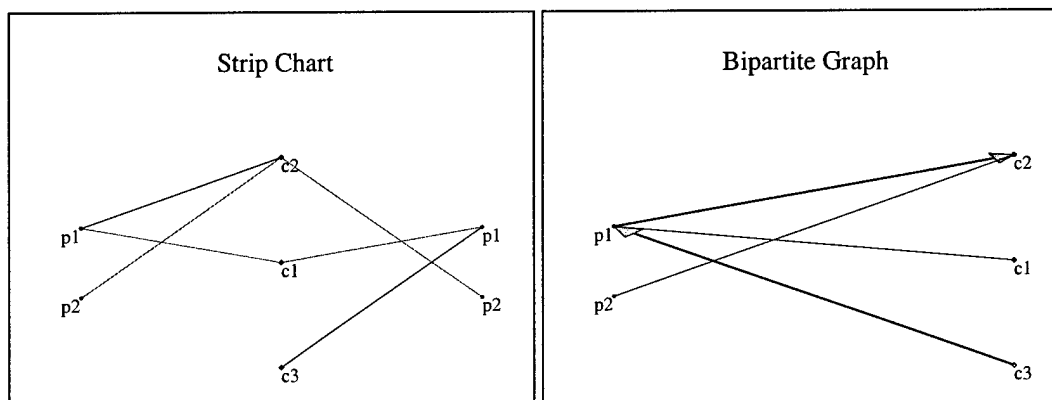
FIGURE 30. The Plots Produced by the Precode_Plot() Procedure

Algorithm C.9. *This procedure displays the synoptic codebook matrix of the precode A based on the order specified by plot_order(A).*

The following are the variables used in this procedure:

E contains the edge list. It has length $NumE$.
$PList, CList$ contain $plot\_order(A)$.
    $PList$ contains the list of plaintext elements in order. It has length $NumP$.
    $CList$ contains the list of codetext elements in order. It has length $NumC$.
$PPoints$ is a table indexed by the plaintext elements which contains the y-coordinate of each vertex.
$CPoints$ is a table indexed by the codetext elements which contains the y-coordinate of each vertex.
yp contains the y-coordinate of the current plaintext vertex.
yc contains the y-coordinate of the current codetext vertex.
$Curves\_Structures$ will contain the CURVES plot structures necessary to display the table lines.
$Text\_Structures$ will contain the TEXT plot structures necessary to display the vertex labels and matrix entries.
$edge$ contains the current edge.
i and j are loop indices.

synoptic_plot:=proc (A) local E, NumE, PList, NumP, CList, NumC,PPoints, CPoints,
edge, yp, yc, Text_Structures,Curves_Structures, i, j;

```
E:=A[_E];
NumE := nops(E);
PList,CList := plot_order(A);
PPoints := table();
```

```
CPoints := table();
NumC := nops(CList);
NumP := nops(PList);

Text_Structures := [];
for i from 1 to NumP do
    PPoints[PList[i]]:=i;
    Text_Structures:=[op(Text_Structures), TEXT([0,i],convert(PList[i],string),
        ALIGNABOVE,ALIGNLEFT,FONT(TIMES,ROMAN,12),COLOR(RGB,0.0,0.0,0.0))];
end do;
for i from 1 to NumC do
    CPoints[CList[i]]:=i;
    Text_Structures:=[op(Text_Structures), TEXT([i,0],convert(CList[i],string),
        ALIGNABOVE,ALIGNLEFT,FONT(TIMES,ROMAN,12),COLOR(RGB,0.0,0.0,0.0))];
end do;

Curves_Structures := [CURVES([[0.5,0.5],[0.5,NumP+0.25]],THICKNESS(1), LINESTYLE(1)),
    CURVES([[0.5,0.5],[NumC+0.25,0.5]],THICKNESS(1), LINESTYLE(1))];
for i from 1 to NumP do
    for j from 1 to NumC do
        yp := PPoints[PList[i]];
        yc := CPoints[CList[j]];
        edge := [PList[i],CList[j]];
        if member(edge, E, 'k') then
            if A[_EData][op(edge)] = [1,1] then
                Text_Structures:=[op(Text_Structures),TEXT([yc,yp],"11",ALIGNABOVE,
                    ALIGNLEFT,FONT(TIMES,ROMAN,12),COLOR(RGB,1.0,0.0,0.0))];
            elif A[_EData][op(edge)] = [1,0] then
                Text_Structures:=[op(Text_Structures),TEXT([yc,yp],"10",ALIGNABOVE,
                    ALIGNLEFT,FONT(TIMES,ROMAN,12),COLOR(RGB,0.0,0.75,0.0))];
            else # A[_EData][op(edge)] = [0,1]
                Text_Structures:=[op(Text_Structures),TEXT([yc,yp],"01", ALIGNABOVE,
                    ALIGNLEFT,FONT(TIMES,ROMAN,12),COLOR(RGB,0.0,0.0,1.0))];
            end if;
        else
            Text_Structures:=[op(Text_Structures),TEXT([yc,yp],"00", ALIGNABOVE,
                ALIGNLEFT,FONT(TIMES,ROMAN,12),COLOR(RGB,0.0,0.0,0.0))];
        end if;
    end do;
end do;
display([PLOT(op(Text_Structures),op(Curves_Structures),AXESSTYLE(NONE))],
    view=[0.0..NumC+0.5,0.0..NumP+0.5]);
end proc:
```

In our example above, the command

```
>  synoptic_plot(A);
```

produces the output in Figure 31.

p1	01	11	10
p2	00	00	11
	c3	c1	c2
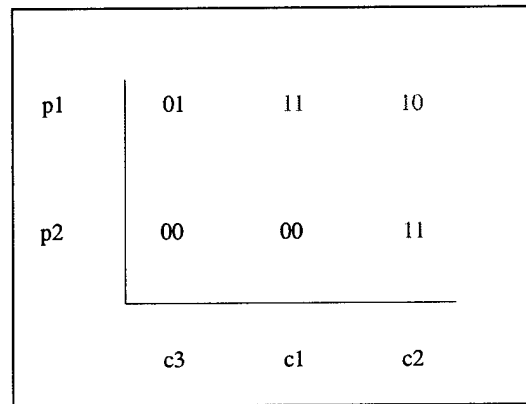
FIGURE 31. The Output Produced by the Synoptic_Plot() Procedure

C.2. The Companion of a Precode. Recall Definition A.5.

Algorithm C.10. *This procedure creates the companion of A. Recall that if $A = (P, C, e, d)$, then its companion is the precode $(P, C, e^{\leftarrow}, d^{\leftarrow})$.*

The following are the variables used in this procedure:

H will contain the companion of A.
edge contains the current edge.
type contains the edge type (i.e. $[1, 0]$ for an encode edge, etc.).
i is a loop index.

cpn:=proc(A) local H,type,edge,i;

```
   #Create the precode.
   new_precode(H):
   add_plain(H,A[_P]); # Add the plaintext set.
   add_code(H,A[_C]); # Add the codetext set.

   # Add the edges.
   for i from 1 to nops(A[_E]) do
      edge := A[_E][i];
      type := A[_EData][op(edge)];
      if type = [1,1] then
         #The edge is of type s.
         add_edge(H,[[edge[1],edge[2],s]]);
      elif type = [1,0] then
         #The edge is of type e.
         add_edge(H,[[edge[1],edge[2],d]]);
      else
         #The edge is of type d.
         add_edge(H,[[edge[1],edge[2],e]]);
      end if;
   end do;
   update_vdata(H);
```

```
    return H;
end proc:
```

In our example above, the command

```
>  precode_plot(cpn(A),"Companion",0);
```

produces the output in Figure 32.
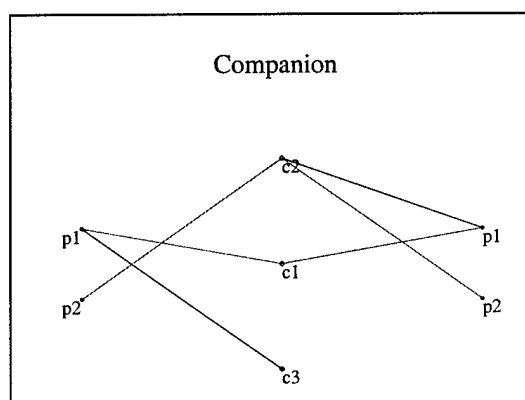
FIGURE 32. The Plot of the Precode Produced by the Cpn() Procedure

C.3. The Opposite of a Precode. Recall Definition A.7.

Algorithm C.11. *This procedure creates the opposite of A. Recall that if A = (P,C,e,d), then its opposite is the precode (C,P,e,d).*

The following are the variables used in this procedure:

H will contain the opposite of A.
edge contains the current edge.
type contains the edge type (i.e. $[1,0]$ for an encode edge, etc.).
i is a loop index.

opp:=proc(A) local H,type,edge,i;

```
    #Create the precode.
    new_precode(H):
    add_plain(H,A[_C]); # Add the plaintext set.
    add_code(H,A[_P]); # Add the codetext set.

    # Add the edges.
    for i from 1 to nops(A[_E]) do
        edge := A[_E][i];
        type := A[_EData][op(edge)];
        if type = [1,1] then
            #The edge is of type s.
            add_edge(H,[[edge[2],edge[1],s]]);
        elif type = [1,0] then
            #The edge is of type e.
```

```
            add_edge(H,[[edge[2],edge[1],d]]);
        else
            #The edge is of type d.
            add_edge(H,[[edge[2],edge[1],e]]);
        end if;
    end do;

    update_vdata(H);
    return H;
end proc:
```

In our example above, the command

```
> precode_plot(opp(A),"Opposite",0);
```

produces the output in Figure 33.
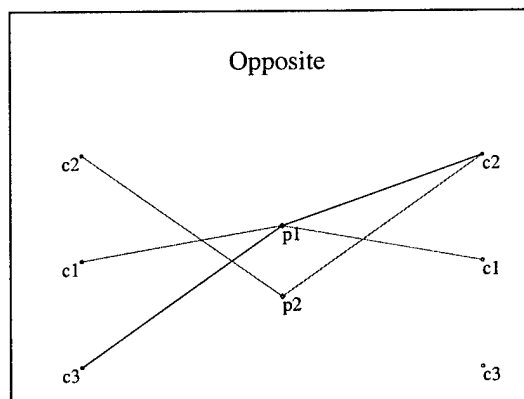
FIGURE 33. The Plot of the Precode Produced by the Opp() Procedure

C.4. The Smash of a Precode. Recall the definition of the "smash" of a precode.

Algorithm C.12. *This procedure is used both to determine if the plaintext elements incident on the codetext element cv need to be "smashed" (when forming the smash of A) and to determine if cv must be split (when forming the split of A). The procedure returns true if cv is a codetext vertex for which there exist distinct plaintext elements p1 and p2 so that $(p1, cv)$ is in the encode relation of A and $(cv, p2)$ is in the decode relation of A. It returns false otherwise.*

The following are the variables used in this procedure:

$Nbrs$ contains the list of vertices ("neighbors") adjacent to cv. It has length $NumNbrs$.
$type\_j$ will contain the type of an edge indexed by j (i.e. $type\_j = A[\_EData][Nbrs[j], cv]$).
$type\_k$ will contain the type of an edge indexed by k (i.e. $type\_k = A[\_EData][Nbrs[k], cv]$).
j and k are loop indices.

```
smash_needed:=proc(A,cv) local Nbrs,NumNbrs,j,k,type_j,type_k;

    Nbrs:=A[_CNbrs][cv];
    NumNbrs := nops(Nbrs);
```

```
# We check to see if cv is a codetext element which is adjacent to at least 2 plaintext
# vertices.
if member(cv,A[_C]) and NumNbrs > 1 then
    # We choose one of the adjacent vertices.
    for j from 1 to NumNbrs do
        # We choose another of the adjacent vertices.
        for k from j+1 to NumNbrs do
            # We now see if there are edges connecting
            # these two vertices "through" cv.
            if member([Nbrs[j],cv],A[_E]) and member([Nbrs[k],cv],A[_E]) then
                type_j := A[_EData][Nbrs[j],cv];
                type_k := A[_EData][Nbrs[k],cv];
                # If one of the edges is undirected, we must return true.
                if (type_j=[1,1]) or (type_k=[1,1]) then
                    return true;
                else
                    # If the edges match up properly, we must return true.
                    if (type_j=[1,0] and type_k=[0,1]) or (type_j=[0,1] and type_k=[1,0]) then
                        return true;
                    end if;
                end if;
            end if;
        end do;
    end do;
end if;
return false;
end proc:
```

Algorithm C.13. *This procedure returns the precode formed from A by "smashing" (i.e. identifying) the plaintext elements incident on the codetext vertex cv.*

The following are the variables used in this procedure:

H is the returned precode.
Nbrs contains the list of vertices ("neighbors") adjacent to *cv*. It has length *NumNbrs*.
NewP contains the list of plaintext elements for *H*.
edge contains the current edge;
type is the *edge* type.
p is the plaintext vertex in *edge*, and *c* is the codetext vertex in *edge*.
i and *j* are loop indices.

```
smash_c:=proc(A,cv) local H,Nbrs,NumNbrs,edge,type,p,c,i,j,NewP;

    Nbrs:=A[_CNbrs][cv];
    NumNbrs := nops(Nbrs);

    new_precode(H): #Create the precode.
    #We first add all the codetext vertices in A to H.
    add_code(H,A[_C]);

    #We remove all the vertices in Nbrs from A[_P] except the first.
    NewP := A[_P];
```

```
    for i from 2 to NumNbrs do
        member(Nbrs[i], NewP, 'k');
        NewP := subsop(k=NULL,NewP);
    end do;
    add_plain(H,NewP);

    #We now add the necessary edges to H.
    for i from 1 to nops(A[_E]) do
        edge := A[_E][i];
        p := edge[1];
        c := edge[2];
        type := A[_EData][p,c];

        # If p is part of the plaintext set we're smashing (the neighbors of cv),
        # then we must replace p with the representative (first) plaintext
        # element in the set.
        for j from 1 to NumNbrs do
            if p = Nbrs[j] then
                p := Nbrs[1];
            end if;
        end do;
        #We need to see if the edge is directed or undirected.
        if type = [1,1] then
            add_edge(H,[[p,c,s]]):
        elif type = [1,0] then
            add_edge(H,[[p,c,e]]):
        else
            add_edge(H,[[p,c,d]]):
        end if;
    end do;
    update_vdata(H);
    return H;
end proc:
```

Algorithm C.14. *This procedure creates the smash of the precode A. Recall that the smash of A is the precode whose vertices are the equivalence classes of an equivalence relation E on P, and its edges are induced by the edges in A. The relation E is defined to be the smallest equivalence relation containing the relation R, where R is the relation on P defined so that $(p1,p2)$ is in R if and only if there exists some cv in C such that p1 and p2 are adjacent (neighbors) of cv and smash_needed(A,cv) is true. Note that the smash of A has the same codetext set as A.*

The following are the variables used in this procedure:

S is a temporary variable which will ultimately contain the smash of A.
N is a table indexed by the vertices of S which contains the vertices which are the "neighbors" of each vertex.
Nbrs contains the list of vertices ("neighbors") adjacent to the codetext element $c[i]$.
    It has length *NumNbrs*.
c is a codetext element.
i is a loop index.

smash:=proc(A) local S,c,Nbrs,NumNbrs,N,i;

```
S:=copy_precode(A);
N:=S[_CNbrs];
# We construct S iteratively by "smashing" the plaintext elements incident on each
# codetext vertex for which smash_needed() returns true.
for i from 1 to nops(S[_C]) do
    c := S[_C][i];
    Nbrs := N[c];
    NumNbrs := nops(Nbrs);

    # We check to see if c has at least two plaintext elements incident on it.
    # If not, no smashing will be necessary.
    if NumNbrs > 1 then
        if smash_needed(S,c) then
            S:=smash_c(S,c);
            N:=S[_CNbrs];
        end if;
    end if;
end do;

return S;
end proc:
```

In our example above, the command

```
> precode_plot(smash(A),"Smash",0);
```

produces the output in Figure 34.
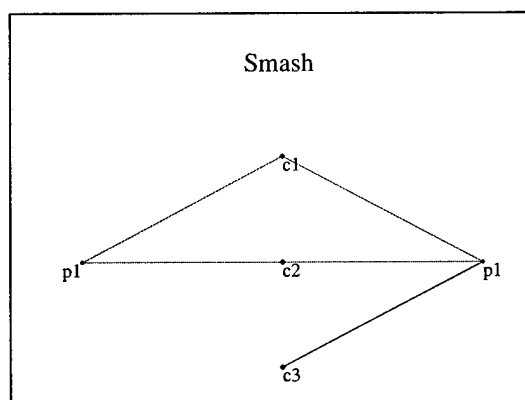
FIGURE 34. The Plot of the Precode Produced by the Smash() Procedure

C.5. **The Split of a Precode.** Recall Algorithms 7.2 and 7.19.

Algorithm C.15. *This procedure returns the precode formed from A by "splitting" the codetext element cv. The user has the option of splitting in either the usual or the "bald" sense. If the parameter bald is true, we split in the bald sense.*

The following are the variables used in this procedure:

H is the returned precode.

E contains the edge list of A. It has length $NumE$.

P contains the set of plaintext elements. It has length $NumP$.

C contains the set of codetext elements. It has length $NumC$.

N is a table indexed by the vertices of A which contains the vertices which are the "neighbors" of each vertex.

$Nbrs$ contains the list of vertices ("neighbors") adjacent to the codetext element $c1$.
    It has length $NumNbrs$.

$edge$ contains the current edge.

$type$ is the $edge$ type (i.e. [1,0] for an encode edge, etc.).

p is the plaintext element and c is the codetext element in $edge$.

$PData$ is a list of the vertices incident on cv.

$EData$ is a table indexed by the elements in $PData$ such that $EData[p]$ contains the edges incident on p and cv.

$PType$ is a list such that $PType[i]$ is the type of the vertex $PData[i]$.
    If there is only an encode edge incident on $PData[i]$, then $PData[i]$ is of type $[1,0]$.
    If there is only a decode edge incident on $PData[i]$, then $PData[i]$ is of type $[0,1]$.
    If there are both, then $PData[i]$ is of type $[1,1]$.

$Type1$ is true if either $PType[i] = [1,0]$ or $Type3 > 1$. It is false otherwise.

$Type2$ is true if either $PType[i] = [0,1]$ or $Type3 > 1$. It is false otherwise.

$Type3$ is the number of i for which $PType[i] = [1,1]$.

i, j, and k are loop indices.

```
split_c:=proc(A,cv,bald) local H,N,Nbrs,NumNbrs,edge,p,c,type,EData,PData,PType,
Type1,Type2,Type3, i,j,k;

    new_precode(H):
    N:=A[_Nbrs];
    Nbrs:=N[cv];
    NumNbrs := nops(Nbrs);
    EData := table();
    PData :=[];
    PType := table();

    #We first add all the plaintext elements in A[_P] to H.
    add_plain(H,A[_P]);
    # We also initialize EData.
    for i from 1 to nops(A[_P]) do
        EData[A[_P][i]] := [];
    end do;

    #We add all the codetext elements in A[_C] to H except cv.
    member(cv, A[_C], 'k');
    add_code(H,subsop(k=NULL,A[_C]));

    for i from 1 to nops(A[_E]) do
        edge := A[_E][i];
        p := edge[1];
        c := edge[2];
        type := A[_EData][p,c];

        #if the edge isn't incident on cv, add it to H.
```

```
    if c <> cv then
        if type = [1,1] then
            add_edge(H,[[p,c,s]]):
        elif type = [1,0] then
            add_edge(H,[[p,c,e]]):
        else
            add_edge(H,[[p,c,d]]):
        end if;
    else
        #The edge is incident on cv. Add it to the table, and we'll deal with it later.
        if not member(p, PData) then
            PData := [op(PData),p];
        end if;
        EData[p]:=[op(EData[p]),edge];
    end if;
end do;

for i from 1 to nops(PData) do
    # We now determine what types of edges are incident on cv. If there is only an
    # encode edge incident on PData[i],then PData[i] is of type [1,0]. If there is only
    # a decode edge, then PData[i] is of type [0,1]. If there are both, then PData[i]
    # is of type [1,1].
    PType[i] := [0,0];
    for j from 1 to nops(EData[PData[i]]) do
        edge := EData[PData[i]][j];
        type := A[_EData][op(edge)];
        if type = [1,1] then
            PType[i] := [1,1];
        elif type = [1,0] then
            PType[i][1] := 1;
        else
            PType[i][2] := 1;
        end if;
    end do;
end do;

# We now determine what types of codetext elements we must add to the precode.
# If there is a plaintext element of type [1,0] or if there is more than one of type
# [1,1], we need to add a codetext element of the form cv_in.
# If there is a plaintext element of type [0,1] or if there is more than one of type
# [1,1], we need to add a codetext element of the form cv_out.
Type1 := false;
Type2 := false;
Type3 := 0;
for i from 1 to nops(PData) do
    if PType[i] = [1,0] then
        Type1 := true;
    elif PType[i] = [0,1] then
        Type2 := true;
    else #PType[i] = [1,1]
        Type3 := Type3 + 1;
```

```
        end if;
end do;

if Type1 = true then
    add_code(H,{cat(cv,_in)});
end if;
if Type2 = true then
    add_code(H,{cat(cv,_out)});
end if;
if not bald and Type3 > 1 then
    add_code(H,{cat(cv,_in)});
    add_code(H,{cat(cv,_out)});
end if;

for i from 1 to nops(PData) do
    #We now add the appropriate edges to H.
    p := PData[i];
    for j from 1 to nops(EData[p]) do
        edge := EData[p][j];
        if A[_EData][op(edge)] = [1,1] then
            #The edge is undirected.
            add_code(H,{cat(p,cv)});
            add_edge(H,[[p,cat(p,cv),s]]):
            #If the user wants to split in the usual sense, we must add more edges.
            if not bald then
                if Type1 = true or Type3 > 1 then
                    add_edge(H,[[p,cat(cv,_in),e]]):
                end if;
                if Type2 = true or Type3 > 1 then
                    add_edge(H,[[p,cat(cv,_out),d]]):
                end if;
            end if;
        else
            #The edge is directed.
            if A[_EData][op(edge)] = [1,0] then
                add_edge(H,[[p,cat(cv,_in),e]]):
                if PType[i]=[1,1] then
                    add_code(H,{cat(p,cv)});
                    add_edge(H,[[p,cat(p,cv),e]]):
                end if;
            else
                add_edge(H,[[p,cat(cv,_out),d]]):
                if PType[i]=[1,1] then
                    add_code(H,{cat(p,cv)});
                    add_edge(H,[[p,cat(p,cv),d]]):
                end if;
            end if;
        end if;
    end do;
end do;
update_vdata(H);
```

```
    return H;
end proc:
```

Algorithm C.16. *This procedure creates the split of the precode A. If the parameter bald is true, we form the "bald" split of A. Otherwise, we form the usual split.*

The following are the variables used in this procedure:

S is a temporary variable which will ultimately contain the split of A.
N is a table indexed by the vertices of S which contains the vertices which are the "neighbors" of each vertex.
c is the current codetext element.
$Nbrs$ contains the list of vertices (neighbors) adjacent to c. It has length $NumNbrs$.
i is a loop index.

```
split:=proc(A,bald) local S,c,N,Nbrs,NumNbrs,i;

    S:=copy_precode(A);
    N:=S[_CNbrs];
    # We construct S iteratively by "splitting" each codetext element for which
    # smash_needed() returns true.
    for i from 1 to nops(S[_C]) do
        c := S[_C][i];
        Nbrs := N[c];
        NumNbrs := nops(Nbrs);
        # We check to see if c has at least two plaintext elements incident on it.
        # If not, no splitting will be necessary.
        if NumNbrs > 1 then
            #We will check to see if c needs to be split.
            #Note that the smash_needed procedure does the appropriate check.
            if smash_needed(S,c) then
                #Let's split c.
                S:=split_c(S,c,bald);
                N:=S[_CNbrs];
            end if;
        end if;
    end do;
    return S;
end proc:
```

In our example above, the commands
```
>  precode_plot(split(A,false));
>  precode_plot(split(A,true));
```
produce the output in Figure 35.

Algorithm C.17. *This procedure creates the ed-split of the precode A as in Algorithm 7.1.*

The following are the variables used in this procedure:

S is a temporary variable which will ultimately contain the split of A.
N is a table indexed by the vertices of S which contains the vertices which are the "neighbors" of each vertex.
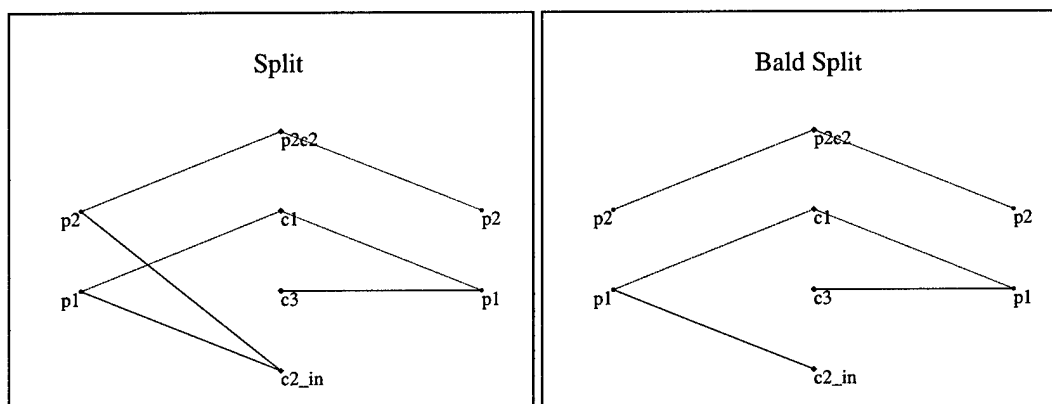
FIGURE 35. The Plots of the Precodes Produced by the Split() Procedure

c is the current codetext element.

Nbrs contains the list of vertices (neighbors) adjacent to the codetext element c.

It has length *NumNbrs*.

i is a loop index.

```
split_ed:=proc(A) local S,c,N,Nbrs,NumNbrs,E_list,D_list,i,j;

    N:=A[_CNbrs];
    new_precode(S);
    add_plain(S,A[_P]);

    # We construct S iteratively by "splitting" each codetext element for which
    # smash_needed() returns true.
    for i from 1 to nops(A[_C]) do
        c := A[_C][i];
        Nbrs := N[c];
        NumNbrs := nops(Nbrs);
        # We will check to see if c needs to be split. Note that the smash_needed procedure
        # does the appropriate check.
        if NumNbrs > 1 and smash_needed(A,c) then
            add_code(S,{cat(c,_in)});
            add_code(S,{cat(c,_out)});
            E_list := [];
            D_list := [];
            for j from 1 to NumNbrs do
                if A[_EData][Nbrs[j],c][1]=1 then
                    E_list := [op(E_list),[Nbrs[j],cat(c,_in),e]];
                end if;
                if A[_EData][Nbrs[j],c][2]=1 then
                    D_list := [op(D_list),[Nbrs[j],cat(c,_out),d]];
                end if;
            end do;
            add_edge(S,E_list);
            add_edge(S,D_list);
        else
```

```
        add_code(S,{c});
        for j from 1 to NumNbrs do
            if A[_EData][Nbrs[j],c]=[1,1] then
                add_edge(S,[[Nbrs[j],c,s]]);
            elif A[_EData][Nbrs[j],c]=[1,0] then
                add_edge(S,[[Nbrs[j],c,e]]);
            else
                add_edge(S,[[Nbrs[j],c,d]]);
            end if;
        end do;
    end if;
  end do;
  update_vdata(S);
  return S;
end proc:
```

In our example above, the command

```
>  precode_plot(split_ed(A),"ED-Split",0);
```

produce the output in Figure 36.
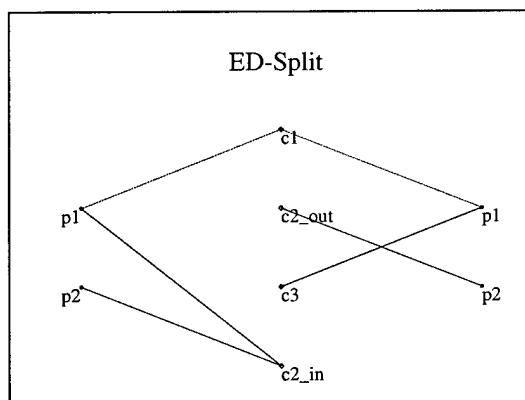
FIGURE 36. The Plot of the Precode Produced by the Split_ED() Procedure

C.6. Parametrizations.

C.6. Parametrizations. Recall the Algorithms given in Section 8.

Algorithm C.18. *This procedure creates the parametrization (F, G) of the precode A given in Algorithm 8.2.*

The following are the variables used in this procedure:

F and G are temporary variables which will ultimately contain the parametrization of A.
$NewC$ will contain the set of codetext elements for F and G.
p is a plaintext element and c is a codetext element.
$type$ is the type of the edge $[p, c]$.
i and j are loop indices.

param_basic:=proc(A) local F,G,NewC,p,c,type,i,j;

```
    new_precode(F);
    new_precode(G);

    # We now construct the plaintext and codetext sets for F and G.
    add_plain(F,A[_P]);
    add_plain(G,A[_C]);
    NewC:={};
    for i from 1 to nops(A[_P]) do
        for j from 1 to nops(A[_C]) do
            NewC := {op(NewC),cat(A[_P][i],A[_C][j])};
        end do;
    end do;
    add_code(F,NewC);
    add_code(G,NewC);

    # We now construct the edge sets for F and G.
    for i from 1 to nops(A[_E]) do
        p := A[_E][i][1];
        c := A[_E][i][2];
        type := A[_EData][p,c];
        if type=[1,1] then
            # [p,c] is of type s.
            add_edge(F,[[p,cat(p,c),s]]);
            add_edge(G,[[c,cat(p,c),s]]);
        elif type=[0,1] then
            # [p,c] is of type d.
            add_edge(F,[[p,cat(p,c),d]]);
            add_edge(G,[[c,cat(p,c),e]]);
        else
            # [p,c] is of type e.
            add_edge(F,[[p,cat(p,c),e]]);
            add_edge(G,[[c,cat(p,c),d]]);
        end if;
    end do;
    update_vdata(F);
    update_vdata(G);
    return F,G;
end proc:
```

Algorithm C.19. *This procedure creates the parametrization (F,G) of the precode A given in Algorithm 8.5.*

The following are the variables used in this procedure:

F and G are temporary variables which will ultimately contain the parametrization of A.
H contains the ED-split of A.
$NewC$ will contain the set of codetext elements for F and G.
edge is the current edge.
type is the *edge* type.
p is a plaintext element and c is a codetext element.

N is $F[\_CNbrs]$.

$Nbrs$ is the list of neighbors of c. It has length $NumNbrs$.

$root$ is the root name of the codetext element; i.e., if $c = a\_out$, then $root = a$.

i and j are loop indices.

```
param_ed_split:=proc(A) local F,G,H,NewC,N,Nbrs,NumNbrs,edge,type,p,c,root,i,j;

    H:=split_ed(A);
    #We construct NewC. It will contain the codetext elements in H which are not isolated.
    #
    NewC:=H[_C];
    for i from 1 to nops(H[_C]) do
        c := H[_C][i];
        if nops(H[_CNbrs][c]) = 0 then
            # Remove c.
            member(c,NewC,'k');
            NewC:=subsop(k=NULL,NewC);
        end if;
    end do;

    #We construct F.
    new_precode(F):
    add_plain(F,A[_P]);
    add_code(F,NewC);
    for i from 1 to nops(H[_E]) do
        edge := H[_E][i];
        p := edge[1];
        c := edge[2];
        type := H[_EData][p,c];
        if type = [1,1] then
            #The edge is of type s.
            add_edge(F,[[p,c,s]]);
        elif type = [1,0] then
            #The edge is of type e.
            add_edge(F,[[p,c,e]]);
        else
            #The edge is of type d.
            add_edge(F,[[p,c,d]]);
        end if;
    end do;
    update_vdata(F);

    #We construct G.
    new_precode(G);
    add_plain(G,A[_C]);
    add_code(G,NewC);

    # Construct the edges for G.
    N := F[_CNbrs];
    for i from 1 to nops(NewC) do
        c := F[_C][i];
```

```
            if length(c) > 3 then
                if substring(c,-3..-1) = _in then
                    root := substring(c,1..-4);
                elif substring(c,-4..-1) = _out then
                    root := substring(c,1..-5);
                else
                    root := c;
                end if;
            else
                root := c;
            end if;
            Nbrs := N[c];
            NumNbrs := nops(Nbrs);
            for j from 1 to NumNbrs do
                type := F[_EData][Nbrs[j],c];
                if type=[1,1] then
                    #[Nbrs[j],c] is of type s.
                    add_edge(G,[[root,c,s]]);
                elif type=[0,1] then
                    #[Nbrs[j],c] is of type d.
                    add_edge(G,[[root,c,e]]);
                else
                    #[Nbrs[j],c] is of type e.
                    add_edge(G,[[root,c,d]]);
                end if;
            end do;
        end do;
        update_vdata(G);
        return F,G;
end proc:
```

Algorithm C.20. *This procedure creates the parametrization (F,G) of the precode A given in Algorithm 8.12.*

The following are the variables used in this procedure:

F and G are temporary variables which will ultimately contain the parametrization of A.
$NewC$ will contain the set of codetext elements for F and G.
edge is a edge.
type is the *edge* type.
p is a plaintext element.
c is a codetext element.
    c_name is the name of an element to be added to $NewC$ which generates either the encode or decode edges incident on c in A.
N is $A[\_CNbrs]$.
$Nbrs$ is the list of neighbors of c. It has length $NumNbrs$.
In_Set contains the set of plaintext elements adjacent to c via an encode edge in A.
Out_Set contains the set of plaintext elements adjacent to c via a decode edge in A.
In_Sets_Table is a table indexed by the elements of $A[\_C]$ such that *In_Sets_Table*$[c]$ contains the name of the element in $NewC$ which induces the encode edges incident on c.
Out_Sets_Table is a table indexed by the elements of $A[\_C]$ such that *Out_Sets_Table*$[c]$ contains the name of the element in $NewC$ which induces the decode edges incident on c.

i and j are loop indices.

```
param_min_ed:=proc(A) local F,G,NewC,In_Set,Out_Set,In_Sets_Table, Out_Sets_Table,p,c,c_name,
N,Nbrs,NumNbrs,edge,type,i,j;

    N := A[_CNbrs];
    In_Sets_Table := table();
    Out_Sets_Table := table();

    # Initialize the plaintext sets for F and G.
    new_precode(F);
    new_precode(G);
    add_plain(F,A[_P]);
    add_plain(G,A[_C]);

    #We construct NewC, the set of codetext elements for F and G.
    NewC:={};
    for i from 1 to nops(A[_C]) do
        c := A[_C][i];

        #We now construct the In_Set and Out_Set for c.
        In_Set := {};
        Out_Set := {};
        Nbrs := N[c];
        NumNbrs := nops(Nbrs);
        for j from 1 to NumNbrs do
            type := A[_EData][Nbrs[j],c];
            if type[1]=1 then
                #We have an encode edge.
                In_Set := {op(In_Set),Nbrs[j]};
            end if;
            if type[2]=1 then
                #We have a decode edge.
                Out_Set := {op(Out_Set),Nbrs[j]};
            end if;
        end do;

        # If In_Set is not empty, we add i_(In_Set) to NewC and set In_Sets_Table[c] = i_(In_Set).
        if nops(In_Set) > 0 then
            c_name := cat(i_,convert(In_Set,symbol));
            NewC := {op(NewC),c_name};
            In_Sets_Table[c]:=c_name;
        end if;
        # If Out_Set is not empty, we add o_(Out_Set) to NewC and set
        # Out_Sets_Table[c] = o_(Out_Set).
        if nops(Out_Set) > 0 then
            c_name := cat(o_,convert(Out_Set,symbol));
            NewC := {op(NewC),c_name};
            Out_Sets_Table[c]:=c_name;
        end if;
    end do;
```

```
          add_code(F,NewC);
          add_code(G,NewC);

          #We add the edges to F and G.
          for i from 1 to nops(A[_E]) do
              edge := A[_E][i];
              p := edge[1];
              c := edge[2];
              type := A[_EData][p,c];
              if type[1] = 1 then
                  # (p,c) is an encode edge in A.
                  add_edge(F,[[p,In_Sets_Table[c],e]]);
                  add_edge(G,[[c,In_Sets_Table[c],d]]);
              end if;
              if type[2] = 1 then
                  # (c,p) is a decode edge in A.
                  add_edge(F,[[p,Out_Sets_Table[c],d]]);
                  add_edge(G,[[c,Out_Sets_Table[c],e]]);
              end if;
          end do;
          update_vdata(F);
          update_vdata(G);
          return F,G;
      end proc:
```

Algorithm C.21. *This procedure creates the parametrization (F, G) of the precode A given in Algorithm 8.22.*

The following are the variables used in this procedure:

F and G are temporary variables which will ultimately contain the parametrization of A.
$NewC$ will contain the set of codetext elements for F and G.
edge is an edge.
type is the *edge* type.
p is a plaintext element.
c is a codetext element.
N is $A[\_CNbrs]$.
$Nbrs$ is the list of neighbors of c. It has length $NumNbrs$.
In contains the set of plaintext elements adjacent to c via an encode edge in A.
Out contains the set of plaintext elements adjacent to c via a decode edge in A.
In_Set is a table indexed by the elements of $A[\_C]$ such that $In\_Set[c]$ contains the set In for c.
Out_Set is a table indexed by the elements of $A[\_C]$ such that $Out\_Set[c]$ contains the set Out for c.
$Set\_Of\_In\_Sets$ is a set of sets. It contains all of the values that In assumes.
$Set\_Of\_Out\_Sets$ is a set of sets. It contains all of the values that Out assumes.
In_Sets_Table is a table indexed by the elements in $Set\_Of\_In\_Sets$.
    $In\_Sets\_Table[In]$ contains the list of all the codetext elements c for which $In = In\_Set[c]$.
Out_Sets_Table is a table indexed by the elements in $Set\_Of\_Out\_Sets$.
    $Out\_Sets\_Table[Out]$ contains the list of all the codetext elements c for which $Out = Out\_Set[c]$.
C_In contains $In\_Sets\_Table[In\_Set[c]]$.
C_Out contains $Out\_Sets\_Table[Out\_Set[c]]$.
e_Name is a table indexed by the elements in $A[\_C]$ such that
    $e\_Name[c]$ is the name of the element in $NewC$ which will generate the encode edges incident

on *c* in *A*.

d_Name is a table indexed by the elements in *A[_C]* such that

d_Name[c] is the name of the element in *NewC* which will generate the decode edges incident on *c* in *A*.

out_not_done is a Boolean indicating whether or not *d_Name[c]* has been set.

i and *j* are loop indices.

```
param_min:=proc(A) local F,G,NewC,In,Out,C_In,C_Out,Set_Of_In_Sets,Set_Of_Out_Sets,
In_Set,Out_Set,In_Sets_Table,Out_Sets_Table,e_Name,d_Name,out_not_done,p,c,N,Nbrs,
NumNbrs,edge,type,i,j;

    N := A[_CNbrs];
    In_Set := table();
    Out_Set := table();
    e_Name := table();
    d_Name := table();
    In_Sets_Table := table();
    Out_Sets_Table := table();
    Set_Of_In_Sets := {};
    Set_Of_Out_Sets := {};
    NewC:={};

    # Initialize the plaintext sets for F and G.
    new_precode(F);
    new_precode(G);
    add_plain(F,A[_P]);
    add_plain(G,A[_C]);

    # Construct the codetext set for F and G.
    for i from 1 to nops(A[_C]) do
        c := A[_C][i];
        In := {};
        Out := {};
        Nbrs := N[c];
        NumNbrs := nops(Nbrs);
        for j from 1 to NumNbrs do
            type := A[_EData][Nbrs[j],c];
            if type[1]=1 then
                # (Nbrs[j],c) is an encode edge in A.
                In := {op(In),Nbrs[j]};
            end if;
            if type[2]=1 then
                # (c,Nbrs[j]) is a decode edge in A.
                Out := {op(Out),Nbrs[j]};
            end if;
        end do;

        In_Set[c] := In;
        Out_Set[c] := Out;

        # Add c to In_Sets_Table[In].
```

```
    if member(In, Set_Of_In_Sets) then
        In_Sets_Table[In]:=[op(In_Sets_Table[In]),c];
    else
        Set_Of_In_Sets := {op(Set_Of_In_Sets),In};
        In_Sets_Table[In]:=[c];
    end if;
    # Add c to Out_Sets_Table[Out].
    if member(Out, Set_Of_Out_Sets) then
        Out_Sets_Table[Out]:=[op(Out_Sets_Table[Out]),c];
    else
        Set_Of_Out_Sets := {op(Set_Of_Out_Sets),Out};
        Out_Sets_Table[Out]:=[c];
    end if;
end do;

# We construct the codetext set for F and G.
for i from 1 to nops(A[_C]) do
    c := A[_C][i];
    In := In_Set[c];
    Out := Out_Set[c];
    C_In := In_Sets_Table[In];
    C_Out := Out_Sets_Table[Out];
    out_not_done := true;

    if nops(In) > 0 then
        # In is not empty.
        if nops(In)=1 and In=Out and nops(C_In)=1 and nops(C_Out)=1 then
            # c is a member of L as defined in the algorithm.
            NewC := {op(NewC),c};
            e_Name[c]:=c;
            d_Name[c]:=c;
            out_not_done := false;
        else
            NewC := {op(NewC),cat(i_,convert(In,symbol))};
            e_Name[c]:=cat(i_,convert(In,symbol));
        end if;
    end if;
    if nops(Out) > 0 and out_not_done then
        # Out is not empty and c is not a member of L as defined in the algorithm.
        NewC := {op(NewC),cat(o_,convert(Out,symbol))};
        d_Name[c]:=cat(o_,convert(Out,symbol));
    end if;
end do;
add_code(F,NewC);
add_code(G,NewC);

# Construct the edges for F and G.
for i from 1 to nops(A[_E]) do
    edge := A[_E][i];
    p := edge[1];
    c := edge[2];
```

```
        type := A[_EData][p,c];
        if type[1] = 1 then
            add_edge(F,[[p,e_Name[c],e]]);
            add_edge(G,[[c,e_Name[c],d]]);
        end if;
        if type[2] = 1 then
            add_edge(F,[[p,d_Name[c],d]]);
            add_edge(G,[[c,d_Name[c],e]]);
        end if;
    end do;
    update_vdata(F);
    update_vdata(G);
    return F,G;
end proc:
```

APPENDIX D

MAPLE CODE FOR COUNTING CODES AND PRECODES

This section gives Maple code for counting codes and precodes up to isomorphism. It implements the formulae in Sections 4 and 5. We use the combinat package, so we include it via the following command.

```
> with(combinat);
```

D.1. **Counting Bipartite Graphs up to M-Equivalence.** Recall Definition 3.10.

Algorithm D.1. *This procedure counts the number of (m,n) bipartite graphs up to M-equivalence, where $M = (M[1], ..., M[k])$ is a partition of m. Recall that two (m,n) bipartite graphs are M-equivalent if they differ from each other by a permutation of the form ab, where a is a permutation of m of type M and where b is a permutation of n. Recall also that a is of type M if a is in $S_{M[1]} \times ... \times S_{M[k]}$, where S_j denotes the symmetric group on j symbols.*

For each $1 \le i \le k$, we let $J[i]$ be a list of the partitions of $M[i]$. Each such partition represents one of the possible cycle structures for the elements of $S_{M[i]}$ (the symmetric group on $M[i]$ elements).

We let $J_Z[i]$ be the list of length $|J[i]|$ such that for each $1 \le j \le |J[i]|$, $J_Z[i][j]$ contains the list of length m which represents the cycle structure shared by the permutations of type $J[i][j]$; that is, for each $1 \le q \le m$, $J_Z[i][j][q]$ contains the number of times q appears in the partition $J[i][j]$. For example, if $m = 8$, $M[i] = 7$, and $J[i][j] = [1,1,2,3]$, then $J_Z[i][j] = [2,1,1,0,0,0,0,0]$. Finally, we let $J_N[i]$ be the list of length $|J[i]|$ such that $J_N[i][j]$ is the number of permutations in $S_M[i]$ with cycle structure $J_Z[i][j]$.

Now, note that there are $Q = |J[1]| \cdot |J[2]| \cdots |J[k]|$ possible cycle structures for the elements in $S_{M[1]} \times ... \times S_{M[k]}$. We let L_Z and L_N be lists of length Q. For each $1 \le j \le Q$, $L_Z[j]$ contains a list of length m which represents one of the possible cycle structures for the elements in $S_{M[1]} \times ... \times S_{M[k]}$. That is, for each $1 \le q \le m$, $L_Z[j][q]$ contains the number of cycles of length q in the corresponding permutation. Each entry in L_N contains the number of permutations of the type represented by the corresponding entry in L_Z.

We let K be a list of the partitions of n. Each such partition represents one of the possible cycle structures for the elements of S_n. We let K_Z be the list of length $|K|$ such that for each $1 \le j \le |K|$, $K_Z[j]$ contains the list of length n which represents the cycle structure shared by the permutations of type $K[j]$; that is, for each $1 \le q \le n$, $K_Z[j][q]$ contains the number of times q appears in the partition $K[j]$. Finally, we let K_N be the list of length $|K|$ such that $K_N[j]$ is the number of permutations in S_n with cycle structure $K_Z[j]$.

By Corollary 3.17, the number of (m,n) bipartite graphs up to M-equivalence is given by the following formula:

$$\frac{1}{M[1]! \cdot M[2]! \cdots M[k]! \cdot n!} \sum_{1 \le a \le Q} \sum_{1 \le b \le |K|} \left(L_N[a] \cdot K_N[b] \left(\prod_{1 \le r \le m} \prod_{1 \le t \le n} 2^{(r,t) \cdot L_Z[a][r] \cdot K_Z[b][t]} \right) \right)$$

The variables used in this procedure are defined as follows:

$m, n, M, a, b, r, t, J, K, MLen = |M|, KLen = |K|, JNum = J_N, JIndex = J_Z, KNum = K_N$, and $KIndex = K_Z$ are all as defined above.

i, j, k, l, w, and v are loop indices.
hj (which corresponds to $h(j)$ in Harary) is the number of permutations corresponding to the current partition.

x and y are temporary variables.

cnt contains the number of times the current k appears in the current partition.

Exponent contains the exponent $(r,t) \cdot LIndex[a][r] \cdot KIndex[b][t]$ for the current r, t, a, and b.

Factor contains the current partial product in the formula above.

NumGraphs contains the number of graphs counted so far.

bipartite:=proc(m,n,M) local J, K, JNum, LNum, KNum, MLen, KLen,JIndex, LIndex,
    KIndex, i, j, k, l, w, v, a, b, r, t, x, y, hj,cnt, Num, NumGraphs, Exponent, Factor;

```
    # Initialize the J data.
    MLen := nops(M); J := array(1..MLen):
    JNum := array(1..MLen):
    JIndex := array(1..MLen):
    for k from 1 to MLen do
        # J[k] contains the partitions of M[k].
        J[k] := partition(M[k]);
        # Initialize JIndex and JNum with arrays to be filled below.
        JIndex[k] := array(1..nops(J[k]));
        JNum[k] := array(1..nops(J[k]));
    end do;

    # Initialize the K data.
    K:=partition(n);
    KLen := nops(K);
    KNum:=[];
    KIndex:=[];

    # Process the J data.
    for k from 1 to MLen do

        # Loop on the number of partitions of M[k]. Each partition represents a possible
        # cycle decomposition of the permutations in S_M[k]. For example, if M[k]=5,
        # then the partition [1,1,1,2] represents permutations in S_M[k] whose unique
        # cycle decomposition has 3 cycles of length 1 and 1 cycle of length 2.
        for j from 1 to nops(J[k]) do
            # Add a new entry to JIndex[k] of the form [0,...,0] (m zeros).
            # Although JIndex[k] will have at most M[k] nonzero terms, we use m entries so
            # that the length is uniform. This will make the processing we do later easier.
            JIndex[k][j] := [seq(0,i=1..m)];

            # hj will contain the number of permutations in S_M[k] which have cycle
            # decomposition corresponding the partition J[k].
            hj:=1;
            # x contains the number of elements from which we may choose to build the current
            # cycle. It is initially M[k]. It will decrease as we count the number of ways
            # to construct each cycle.
            x:=M[k];

            #Loop once for each entry in the current partition.
            for i from 1 to nops(J[k][j]) do
                y := J[k][j][-i];
```

```
# cnt contains the number of entries in J[k][j] with value y, that is, the number
# of cycles with length y in the cycle decomposition. Recall that -i references
# the ith-to-last element in the list.
cnt:=0;
if i < nops(J[k][j]) then
    #This is not the last entry in the partition.
    for l from i to nops(J[k][j]) while (J[k][j][-l]=y) do
        #There is another entry in J[k][j] with value y.
        cnt:=cnt+1;

        # There are (x choose y) ways to choose the y elements for the current
        # cycle. There are (y-1)! possible cycles containing each choice of y
        # elements ((y-1)! is the number of circular orderings of y elements).
        # For example, if y = 3 (corresponding to a cycle of length 3 in the
        # permutation corresponding to this partition), then there are 2!
        # possible cycles containing the 3 particular elements chosen. Thus,
        # the factor contributed to hj for this cycle is (x choose y)*(y-1)!
        hj:=hj*numbcomb(x,y)*(y-1)!;

        # After we've chosen y elements from {1,...,x}, we must eliminate them
        # from the pool.
        x:= x-y;
    end do;

    if cnt >= 1 then
        # We had multiple cycles of length y, and we've handled them in the above
        # loop, so skip past them.
        i:=l-1;

        # However, we've overcounted by the number of ways of ordering the cycles
        # of the same type. For example, if we had 3 cycles of length 2, we've
        # counted each combination of 3 such cycles 3! times. We must correct
        # this.
        hj:=hj/cnt!;
    end if;
else
    #This is the last entry in the partition.
    hj:=hj*numbcomb(x,J[k][j][1])*(J[k][j][1]-1)!;
end if;

#In some cases, cnt might still be 0. This is incorrect.
JIndex[k][j][y] := max(cnt,1);
end do;

#Add the number of permutations corresponding to this partition to the JNum list.
JNum[k][j] := hj;
end do; # End of loop indexed by j.
end do; # End of loop indexed by k

# We now finish processing the J data by combining the information from each of the
# JIndex[k] into one list LIndex, and similarly for LNum.
```

```
# LIndex and LNum are arrays. We need to treat them as lists.
# We set LIndex := JIndex[1] and LNum := JNum[1]. We'll add the entries
# for k from 2 to MLen below.
LIndex := convert(JIndex[1],listlist);
LNum := convert(JNum[1],listlist);

for k from 2 to MLen do
    # We will use the existing terms in LIndex along with the terms in JIndex[k] to add
    # updated terms to LIndex. We will then remove all of the old LIndex terms from
    # the front of LIndex. We will do similarly for LNum.
    Num := nops(LIndex);
    for w from 1 to Num do
        for v from 1 to nops(J[k]) do
            # LIndex contains N_1*...*N_(k-1) entries, where N_i=nops(J[i]) is the number
            # of partitions of M[k]. Each entry corresponds to a product permutation in
            # S_M[1] x ... x S_M[k-1], which is the product of one permutation
            # corresponding to a partition of M[1], one corresponding to a partition of M[2], etc.
            # Each entry in LIndex contains a list of the number of cycles of each length in
            # the corresponding permutation.

            # J[k][v] contains a partition of M[k], and JIndex[k][v] contains a list of the
            # number of cycles of each length in the permutations represented by J[k][v].

            # The number of cycles of a given length in the product of the permutations
            # represented by LIndex[w] and JIndex[k][v] is the sum of the number of cycles
            # of that length in each of those permutations.
            # (Note: Here is where it pays to let those lists all be of length m.)
            LIndex := [op(LIndex),[seq(LIndex[w][i]+JIndex[k][v][i],i=1..m)]];

            # The number of product permutations that can be formed using one permutation
            # of the type specified in LIndex[w] and one permutation of the type
            # specified in JIndex[k][v] is the product of the number of permutations of
            # type LIndex[w] and the number of permutations of type JIndex[k][v].
            LNum := [op(LNum),(LNum[w]*JNum[k][v])];
        end do;
    end do;

    # We now strip the old terms from LIndex and LNum.
    LIndex := subsop(seq(i=NULL,i=1..Num),LIndex);
    LNum := subsop(seq(i=NULL,i=1..Num),LNum);
end do;

# We now process the K data. (See the above loop for comments.)
# Loop on the number of partitions of n.
for j from 1 to KLen do
    #Add a new entry to KIndex of the form [0,...,0]
    KIndex := [op(KIndex),[seq(0,i=1..n)]];
    hj:=1;
    x:=n;
```

```
#Loop once for each entry in the current partition
for i from 1 to nops(K[j]) do
    cnt:=0;
    if i < nops(K[j]) then
        #This is not the last entry in the partition.
        for l from i to nops(K[j]) while (K[j][-l] = K[j][-i]) do
            cnt:=cnt+1;
            hj:=hj*numbcomb(x,K[j][-l])*(K[j][-l]-1)!;
            x:= x-K[j][-i];
        end do;

        if cnt >= 1 then
            i:=l-1;
            hj:=hj/cnt!;
        end if;
    else
        #This is the last entry in the partition.
        hj:=hj*numbcomb(x,K[j][1])*(K[j][1]-1)!;
    end if;
    KIndex[j][K[j][-i]] := max(cnt,1);
end do;

KNum := [op(KNum),hj];
end do;

#This loop computes the number given by the formula using the data collected above.
NumGraphs := 0;

#This loop corresponds to Alpha in the formula.
for a from 1 to nops(LIndex) do
    #This loop corresponds to Beta in the formula.
    for b from 1 to KLen do
        Factor:=1;
        #This loop corresponds to r in the formula.
        for r from 1 to m do
            #This loop corresponds to t in the formula.
            for t from 1 to n do
                #Compute the exponent for this r and t.
                Exponent:=gcd(r,t)*LIndex[a][r]*KIndex[b][t];

                #Update the partial product.
                Factor:=Factor*(2^Exponent);
            end do;
        end do;

        #Update the partial sum.
        NumGraphs:=NumGraphs+(LNum[a]*KNum[b]*Factor);
    end do;
end do;

#Scale as in the formula.
```

```
    NumGraphs := NumGraphs/(mul(M[k]!,k=1..nops(M))*n!);
end proc:
```

Algorithm D.2. *This procedure counts the number of (m, n) bipartite graphs for which NONE of the n vertices of the second color are isolated up to M-equivalence, where M is a partition of m (see the comments in bipartite(m, n, M) for the definition of M-equivalence). Then number of such graphs is given by the following formula in Corollary 3.19:*

$$bipartite\_strict(m, n, M) = bipartite(m, n, M) - \sum_{1 \leq k \leq n} bipartite\_strict(m, (n - k), M)$$

The variables used in this procedure are defined as follows:

k is a loop index
NumGraphs contains the number of graphs counted so far.

bipartite_strict:=proc(m,n,M) local k, NumGraphs;

```
    if n = 0 then
        # bipartite_strict(m,0,M) = 1 (the graph with no edges).
        NumGraphs := 1;
    else
        #bipartite_strict(m,n,M)=bipartite(m,n,M)-sum_{1 <= k <= n}bipartite_strict(m,(n-k),M).
        #Since bipartite_strict(m,0,M) = 1, we let k run to n-1 and then subtract 1.
        NumGraphs := bipartite(m,n,M)-add(bipartite_strict(m,n-k,M),k=1..(n-1));
        # Now subtract off the code with no edges.
        NumGraphs := NumGraphs - 1;
    end if;

    NumGraphs;
end proc:
```

Algorithm D.3. *This procedure counts the number of (m, n) bipartite graphs with no isolated vertices (that is, (m, n) bipartite graphs which are both 1-strict and 2-strict) up to isomorphism. Let $B(m, n)$ represent Bipartite_Strict(m, n). Then, as in Theorem 3.20, the formula is*

$$B(m, n) = bipartite(m, n) - \sum_{1 \leq j \leq (m-1)} \sum_{1 \leq k \leq (n-1)} B(j, k) - \sum_{1 \leq j \leq (m-1)} B(j, n) - \sum_{1 \leq k \leq (n-1)} B(m, k),$$

where $B(m, 0) = 0 = B(0, n)$.

The variables used in this procedure are defined as follows:

NumGraphs contains the number of graphs counted so far.

Bipartite_Strict:=proc(m,n) local NumGraphs;

```
    #If m = 0 or n = 0, Bipartite_Strict(m,n) = 0.
    if m = 0 or n = 0 then
        NumGraphs := 0;
    else
```

```
        NumGraphs := bipartite(m,n,[m])
            - add(add(Bipartite_Strict(j,k),k=1..(n-1)), j=1..(m-1))
            - add(Bipartite_Strict(j,n), j=1..(m-1))
            - add(Bipartite_Strict(m,k), k=1..(n-1));
    end if;
    NumGraphs;
end proc:
```

D.2. Counting Mixed Bipartite Graphs up to Isomorphism. Recall Definition 3.21.

Algorithm D.4. *This procedure counts the number of (m,n) mixed bipartite graphs up to isomorphism.*

Let $A = S_m \times \cdots \times S_n$; let $j_r(a)$ denote the number of cycles of length r in the cycle decomposition of a in S_m; and let $j_t(b)$ denote the number of cycles of length t in the cycle decomposition of b in S_n. Then the number of (m,n) mixed bipartite graphs up to isomorphism is given by the following formula in Theorem 3.23 and Notation 3.24:

$$\frac{1}{m!n!} \sum_{(a,b)\in A} \prod_{1\leq r\leq m; 1\leq t\leq n} 2^{(2\cdot(r,t)\cdot j_r(a)\cdot j_t(b))}$$

The variables used in this procedure are defined as follows:

m and n are parameters corresponding to the m and n above.

a, b, r, and t are loop indices corresponding to the a, b, r and t in the above formula.

i, j, k, and l are loop indices.

hj (which corresponds to h(j) in Harary) is the number of permutations corresponding to the current partition.

Exponent contains the exponent $((r,t)J(r)j\_t(b))$ for the current r, t, a, and b.

Factor contains the current partial product in the formula above.

J contains a list of the partitions of m.

$JLen$ is the number of partitions of m.

$JNum$ is a list such that $JNum[j]$ is the number of permutations in S_m corresponding to the partition $J[j]$ of n.

$JIndex$ is a list such that $JIndex[j]$ contains the list of length m such that $JIndex[j][k]$ contains the number of times k appears in the partition $J[j]$.

For example, if $m = 7$ and $J[j] = [1,1,2,3]$, then $JIndex[j] = [2,1,1,0,0,0,0]$.

K contains a list of the partitions of n.

$KLen$ is the number of partitions of n.

$KNum$ is a list such that $KNum[j]$ is the number of permutations in S_n corresponding to the partition $K[j]$ of n.

$KIndex$ is a list such that $KIndex[j]$ contains the list of length n such that $KIndex[j][k]$ contains the number of times k appears in the partition $K[j]$.

(See the description of $JIndex$ for an example.)

x and y are temporary variables.

cnt contains the number of times the current k appears in the current partition.

(See the definition of $JIndex$).

$NumGraphs$ contains the number of graphs counted so far.

```
mixed_bipartite:=proc(m,n) local J, K, JNum, KNum, JLen, KLen,
    JIndex, KIndex, i, j, l, a, b, r, t, x, y, hj, cnt, NumGraphs,
    Exponent, Factor;
```

```
# Initialize the J data.
J:=partition(m);
JLen := nops(J);
JNum:=[];
JIndex:=[];

# Initialize the K data.
K:=partition(n);
KLen := nops(K);
KNum:=[];
KIndex:=[];

# Process the J data.
# Loop on the number of partitions of m.
for j from 1 to JLen do
    # Add a new entry to JIndex of the form [0,...,0] (m zeros).
    JIndex := [op(JIndex),[seq(0,i=1..m)]];

    # hj will contain the number of permutations in S_m which have cycle decomposition
    # corresponding to the partition J[j].
    hj:=1;
    # x contains the number of elements from which we may choose to build the current
    # cycle. It is initially m. It will decrease as we count the number of ways to construct
    # each cycle.
    x:=m;

    #Loop once for each entry in the current partition
    for i from 1 to nops(J[j]) do
        y := J[j][-i];

        # cnt contains the number of entries in J[j] with value y, that is, the number of
        # cycles with length y in the cycle decomposition. Recall that -i references the
        # ith-to-last element in the list.
        cnt:=0;
        if i < nops(J[j]) then
            #This is not the last entry in the partition.
            for l from i to nops(J[j]) while (J[j][-l] = J[j][-i]) do
                #There is another entry in J[j] with value y.
                cnt:=cnt+1;

                # There are (x choose y) ways to choose the y elements for the current cycle.
                # There are (y-1)! possible cycles containing each choice of y elements since
                # (y-1)! is the number of circular orderings of y elements.
                # For example, if y = 3 (corresponding to a cycle of length 3 in the permutation
                # corresponding to this partition), then there are 2! possible cycles containing
                # the 3 particular elements chosen. Thus, the factor contributed to hj for this
                # cycle is (x choose y)*(y-1)!
                hj:=hj*numbcomb(x,y)*(y-1)!;

                # After we've chosen y elements from {1,...,x}, we must eliminate them
```

```
            # from the pool.
            x:= x-y;
        end do;

    if cnt >= 1 then
            # We had multiple cycles of length y, and we've handled them in the above loop,
            # so skip past them.
            i:=l-1;

            # However, we've overcounted by the number of ways of ordering the cycles of
            # the same type. For example, if we had 3 cycles of length 2, we've counted each
            # combination of 3 such cycles 3! times. We must correct this.
            hj:=hj/cnt!;
        end if;
    else
        #This is the last entry in the partition.
        hj:=hj*numbcomb(x,J[j][1])*(J[j][1]-1)!;
    end if;
    JIndex[j][y] := max(cnt,1);
    end do;

    #Add the number of permutations corresponding to this partition to the JNum list.
    JNum := [op(JNum),hj];
end do;


# We now process the K data. (See the above loop for comments.)
# Loop on the number of partitions of n.
for j from 1 to KLen do
    #Add a new entry to KIndex of the form [0,...,0]
    KIndex := [op(KIndex),[seq(0,i=1..n)]];
    hj:=1;
    x:=n;

    #Loop once for each entry in the current partition
    for i from 1 to nops(K[j]) do
        y := K[j][-i];
        cnt:=0;
        if i < nops(K[j]) then
            #This is not the last entry in the partition.
            for l from i to nops(K[j]) while (K[j][-l] = y) do
                cnt:=cnt+1;
                hj:=hj*numbcomb(x,y)*(y-1)!;
                x:= x-y;
            end do;

            if cnt >= 1 then
                i:=l-1;
                hj:=hj/cnt!;
            end if;
        else
            #This is the last entry in the partition.
```

```
            hj:=hj*numbcomb(x,K[j][1])*(K[j][1]-1)!;
        end if;
        KIndex[j][y] := max(cnt,1);
     end do;


  KNum := [op(KNum),hj];
end do;


# This loop computes the number given by the formula using the data collected above.
NumGraphs := 0;
# This loop corresponds to Alpha in the formula.
for a from 1 to JLen do
   # This loop corresponds to Beta in the formula.
   for b from 1 to KLen do
       Factor:=1;
       # This loop corresponds to r in the formula.
       for r from 1 to m do
          # This loop corresponds to t in the formula.
          for t from 1 to n do
             # Compute the exponent for this r and t.
             Exponent:=2*gcd(r,t)*JIndex[a][r]*KIndex[b][t];
             # Update the partial product.
             Factor:=Factor*(2^Exponent);
          end do;
       end do;


       # Update the partial sum.
       NumGraphs:=NumGraphs+(JNum[a]*KNum[b]*Factor);
   end do;
end do;


# Scale as in the formula.
NumGraphs := NumGraphs/(m!*n!);
end proc:
```

D.3. Counting S Codes. Recall Definition 4.2.

Algorithm D.5. *This procedure counts the number of (m, n) strictly S codes. By Lemma 4.5, this is simply the number of partitions of n into m nonnegative parts; that is, the number of partitions of n into m or fewer parts.*

The variables used in this procedure are defined as follows:

J contains a list of the partitions of n.
j is a loop index.
NumCodes contains the number of strictly S codes counted so far.

```
s_strict:=proc(m,n) local J,j,NumCodes;

  J:=partition(n);

  NumCodes:=0;
```

```
# Loop on the number of partitions of n.
for j from 1 to nops(J) do
    if nops(J[j]) <= m then
        NumCodes := NumCodes + 1;
    end if
end do;
NumCodes;
end proc:
```

Algorithm D.6. *This procedure counts the number of (m,n) S codes. By Lemma 4.5,*

$$s(m,n) = \sum_{1 \leq i \leq n} s\_strict(m,i).$$

The variables used in this procedure are defined as follows:

i is a loop index.
NumCodes contains the number of strictly S codes counted so far.

```
s:=proc(m,n) local i,NumCodes;

    NumCodes:=0;

    # i contains the number of o columns present.
    for i from 1 to n do
        NumCodes := NumCodes + s_strict(m,i);
    end do;
    NumCodes;
end proc:
```

D.4. Counting E and D Codes. Recall Definition 4.2.

Algorithm D.7. *This procedure counts the number of (m,n) E (or D) codes. As in Lemma 4.6 this is the number of (m,n) bipartite graphs minus 1.*

The variables used in this procedure are defined as follows:

NumCodes contains the number of E codes.

```
e:=proc(m,n) local NumCodes;

    NumCodes := bipartite(m,n,[m]) - 1;

end proc:
```

Algorithm D.8. *This procedure counts the number of (m,n) strictly E (or D) codes. As in Lemma 4.6 this is the number of (m,n) 2-strict bipartite graphs.*

The variables used in this procedure are defined as follows:

NumCodes contains the number of strictly E codes.

e_strict:=proc(m,n) local NumCodes;

NumCodes := bipartite_strict(m,n,[m]);

end proc:

D.5. Counting SE and SD Codes. Recall Definition 4.8.

Algorithm D.9. *This procedure counts the number of (m,n) SE (or SD) codes. By Lemma 4.13,*

$$se(m,n) = 0 \ \text{if } n < 2$$

and

$$se(m,n) = \sum_{1 \leq c \leq (n-1)} \sum_{J} \left(bipartite\,(m,c,K(J)) - 1 \right)$$

for $n \geq 2$, where the second sum is over all partitions J of $s = n - c$ such that $|J| \leq m$ and where $K(J)$ is the partition of m formed by tacking on at most one term to the partition $M(J)$ of $|J|$ which tracks the number of times each value in J appears.

The variables used in this procedure are defined as follows:

fc is the number of "free" codetext elements; i.e., those which aren't of type s.
$s = n - fc$ is the number of s columns in the matrix representation of the code.
J contains a list of the partitions of $n - fc$.
Each partition represents the types of s components in the current code.
 For example, if $J[j] = [1,1,3]$, there are two S components each incident on one codetext element, and one S component incident on three codetext elements.
$p = |J|$ is the number of entries in the current partition of $n - fc$, i.e., the number of connected s components in the current code. In the above example, $p = 3$. We use the letter p for "plaintext", since there is precisely one plaintext element in each s component.
fp is the number of "free" plaintext elements; i.e., those which aren't part of the s components.
K tracks the number of each different type of s component in $J[j]$.
 For $J[j] = [1,1,3]$, $K = [2,1]$, indicating two components of one type and one of another type.
i,j are loop indices.
$Cnt, Factor$, and s are temporary variables.
$NumCodes$ contains the number of SE codes counted so far.

se:=proc(m,n) local J,K,p,fc,fp,i,j,NumCodes,Cnt,Factor,s;

```
    NumCodes := 0;
    if n < 2 then
        return(0);
    end if;

    # fc is the number of "free" columns, i.e., the columns which are not the s columns.
    # They will be e or o columns. Since we need at least one s column, fc can be at most n-1.
    for fc from 1 to n-1 do
        s:=n-fc;
        J:=partition(s);
```

```
#Loop on the number of partitions of s.
for j from 1 to nops(J) do

    # We need to determine the number of each type of S component. For example, if
    # J[j]=[1,1,3], there are two S components each incident on one plaintext element,
    # and one S component incident on three plaintext elements. Thus, we construct
    # K=[2,1] to represent this.

    # p is the number of s components. fp=m-p is the number of "free" plaintext
    # vertices, that is, plaintext vertices which are not part of an s component.
    p := nops(J[j]);
    fp := m-p;

    # There must be one plaintext entry for each entry in J[j] since each entry in
    # J[j] is supposed to represent an s component, and each s component is attached
    # to a distinct plaintext vertex. Thus, we must disregard any partition in J
    # with more than m parts.
    if p <= m then
        # We now construct K for this partition.
        Cnt := 1;
        K := [];
        for i from 2 to p do
            if J[j][i] = J[j][i-1] then
                Cnt := Cnt + 1;
            else
                K := [op(K),Cnt];
                Cnt := 1;
            end if;
        end do;


        K := [op(K),Cnt];

        #We construct a partition of m to send to bipartite().
        if fp <> 0 then
            K:=[op(K),fp]
        end if;

        # We need to be sure to subtract the code with no edges since it represents a
        # code with no e edges.
        NumCodes:=NumCodes+bipartite(m,fc,K)-1;
    end if;
    end do;
end do;
NumCodes;
end proc:
```

Algorithm D.10. *This procedure counts the number of (m,n) strictly SE (or SD) codes. By Lemma 4.13,*

$$se\_strict(m, n) = se(m, n) - \sum_{1 \le k \le (n-1)} se\_strict(m, n - k).$$

The variables used in this procedure are defined as follows:

NumCodes is the number of strictly *SE* codes counted so far.

```
se_strict:=proc(m,n) local i,j,k,l,NumCodes;

    if n = 1 then
        NumCodes := 0;
    else
        NumCodes := se(m,n)-add(se_strict(m,n-k),k=1..(n-1));
    end if;
    NumCodes;
end proc:
```

D.6. Counting SED Codes. Recall Definition 4.8.

Algorithm D.11. *This procedure counts the number of* (m, n) *SED codes. Let* $bi\_s(i, j)$ *denote* $bipartite\_strict(i, j)$. *By Lemma 4.14,*

$$sed(m, n) = 0 \ \text{if} \ n < 3$$

and

$$sed(m, n) = \sum_{1 \le c \le (n-1)} \sum_{J} \sum_{1 \le e \le (n-s-1)} \sum_{1 \le d \le (n-s-e)} (bi\_s\,(m, e, K(J)) \cdot bi\_s\,(m, d, K(J))) \ \text{for} \ n > 3,$$

where the second sum is over all partitions J of $s = n - c$ such that $|J| \le m$ and where $K(J)$ is the partition of m formed by tacking on at most one term to the partition $M(J)$ of $|J|$ which tracks the number of times each value in J appears.

The variables used in this procedure are defined as follows:

fc is the number of "free" codetext elements; i.e., those which aren't of type s.
$s = n - fc$ is the number of s columns in the matrix representation of the code.
J contains a list of the partitions of $n - fc$. Each partition represents the types of s components in the current code. For example, if $J[j] = [1, 1, 3]$, there are two S components each incident on one codetext element, and one S component incident on three codetext elements.
$p = |J|$ is the number of entries in the current partition of $n - fc$, i.e., the number of connected s components in the current code. In the above example, $p = 3$. We use the letter p for "plaintext", since there is precisely one plaintext element in each s component.
fp is the number of "free" plaintext elements; i.e., those which aren't part of the s components.
K tracks the number of each different type of s component in $J[j]$.
    For $J[j] = [1, 1, 3]$, $K = [2, 1]$, indicating two components of one type and one of another type.
i, j, e, and d are loop indices.
$Cnt, Factor$, and s are temporary variables.
NumCodes contains the number of *SED* codes counted so far.

```
sed:=proc(m,n) local J,K,p,fc,fp,i,j,NumCodes,Cnt,Factor,s,e,d;
    NumCodes := 0;
    if n < 3 then
        return(0);
    end if;
    # fc is the number of "free" columns, i.e., the columns which are not the s columns.
    # They will be e, d, or o columns. Since we need at least one s column, fc can be at
    # most n-1.
    for fc from 1 to n-1 do
        s:=n-fc;
        J:=partition(s);
        #Loop on the number of partitions of s.
        for j from 1 to nops(J) do
            # We need to determine the number of each type of S component. For example, if
            # J[j]=[1,1,3], there are two S components each incident on one plaintext element,
            # and one S component incident on three plaintext elements. Thus, we construct
            # K=[2,1] to represent this. p is the number of s components.
            # fp is the number of "free" plaintext vertices, that is, plaintext
            # vertices which are not part of an s component.
            p := nops(J[j]);
            fp := m-p;
            # There must be one plaintext entry for each entry in J[j] since each entry in J[j]
            # is supposed to represent an s component, and each s component is attached to a
            # distinct plaintext vertex. Thus, we must disregard any partition in J with more
            # than m parts.
            if p <= m then
                # We now construct K for this partition.
                Cnt := 1;
                K := [];
                for i from 2 to p do
                    if J[j][i] = J[j][i-1] then
                        Cnt := Cnt + 1;
                    else
                        K := [op(K),Cnt];
                        Cnt := 1;
                    end if;
                end do;

                K := [op(K),Cnt];
                #We construct a partition of m to send to bipartite().
                if fp <> 0 then
                    K:=[op(K),fp]
                end if;
                # We loop on the number of e vertices. We need at least one e vertex.
                # Since we must have at least one d vertex, e can be at most n-(s+1).
                for e from 1 to n-s-1 do
                    # We loop on the number of d vertices. We need at least one d vertex.
                    # The remaining n-(s+e+d) vertices are the o columns. We only need to loop for
                    # d <= e since for d > e, the number of sed codes with d codetext elements of
                    # type "d" and e of type "e" is the same as the number with d elements of
```

```
            # type "e" and e elements of type "d" (by symmetry). Thus, if d <> e, we
            # just need to double the count.
            for d from 1 to n-s-e while d <= e do
                if d = e then
                    NumCodes := NumCodes + (bipartite_strict(m,e,K)^2);
                else
                    NumCodes := NumCodes +(2*bipartite_strict(m,e,K)*bipartite_strict(m,d,K));
                end if;
            end do;
        end do;
    end if;
  end do;
 end do;
 NumCodes;
end proc:
```

D.7. Counting ED Codes. Recall Definition 4.8.

Algorithm D.12. *This procedure counts the number of (m,n) ED codes. By Lemma 4.11,*

$$ed(m,n) = 0 \ \text{if } n < 2$$

and

$$ed(m,n) = \sum_{0 \le j \le (n-2)} \sum_{1 \le k \le (n-j-1)} (e\_strict\,(m,k) \cdot e\_strict\,(m,n-j-k)) \ \ \text{if } n \ge 2.$$

The variables used in this procedure are defined as follows:

i and j are loop indices
NumCodes is the number of *ED* codes counted so far.

```
ed:=proc(m,n) local i,j,NumCodes;

    NumCodes := 0;
    # j will run through the number of o columns. This will be at most n-2 since we require
    # at least one d and one e column.
    for j from 0 to n-2 do
        # i will run through the number of e columns. Since there are j o columns, there can
        # be at most n-(j+1) e columns.
        for i from 1 to n-j-1 do
            # We add the number of codes with i e columns and n-j-i d columns. Since i is the
            # actual number of e columns, we must use e_count instead of e_full_count.
            # We must similarly use e_count for the d columns as well.
            NumCodes := NumCodes + (e_strict(m,i)*e_strict(m,n-j-i));
        end do;
    end do;
    NumCodes;
end proc:
```

D.8. **Counting All Codes.** We now put it all together.

Algorithm D.13. *This procedure counts the number of* (m, n) *codes up to isomorphism. By Theorem 4.10,*

$$code(m, n) = o(m, n) + s(m, n) + 2 \cdot e(m, n) + 2 \cdot se(m, n) + ed(m, n) + sed(m, n).$$

The variables used in this procedure are defined as follows:

$O = o(m, n)$
$S = s(m, n)$
$E = e(m, n)$
$SE = se(m, n)$
$ED = ed(m, n)$
$SED = sed(m, n)$
$NumCodes$ is the number of codes counted so far.

```
code:=proc(m,n,printflag) local NumCodes,O,E,S,SE,ED,SED;

    #If m = 0 or n = 0, there is only one code.
    if m = 0 or n = 0 then
        if (printflag = true) then
            printf("%s %d\ n","The total is",1);
        end if;
        return(1);
    end if;

    # There is only one O code–the code with no edges.
    O := 1;
    if (printflag = true) then
        printf("%s %d\ n", "O is ", O);
    end if;

    S := s(m,n);
    if (printflag = true) then
        printf("%s %d\ n", "S is ", S);
    end if;

    # E is also the number of D codes.
    E := e(m,n);
    if (printflag = true) then
        printf("%s %d\ n", "E=D is ", E);
    end if;

    # SE is also the number of SD codes.
    SE := se(m,n);
    if (printflag = true) then
        printf("%s %d\ n", "SE=SD is ", SE);
    end if;

    ED := ed(m,n);
    if (printflag = true) then
```

```
        printf("%s %d\ n", "ED is ", ED);
    end if;

    SED := sed(m,n);
    if (printflag = true) then
        printf("%s %d\ n", "SED is ", SED);
    end if;

    NumCodes := O + S + 2*E + 2*SE + ED + SED;
    if (printflag = true) then
        printf("%s %d\ n", "The total is", NumCodes);
    else
        NumCodes;
    end if;
end proc:
```

D.9. Computing Compositions. The following is needed to count janiform codes.

Algorithm D.14. *This procedure computes all of the 4-compositions of n. It returns them in a list of lists C. Let X denote a nonzero entry and 0 represent a zero entry. Then C satisfies the following conditions:*

$C[1]$ contains the compositions of the form $[X, X, X, X]$ and $[X, X, X, 0]$.
$C[2]$ contains the compositions of the form $[X, X, 0, X]$ and $[X, X, 0, 0]$.
$C[3]$ contains the compositions of the form $[X, 0, X, X]$ and $[X, 0, X, 0]$.
$C[4]$ contains the compositions of the form $[0, X, X, X]$ and $[0, X, X, 0]$.
$C[5]$ contains the compositions of the form $[X, 0, 0, X]$ and $[X, 0, 0, 0]$.
$C[6]$ contains the compositions of the form $[0, X, 0, X]$ and $[0, X, 0, 0]$.
$C[7]$ contains the compositions of the form $[0, 0, X, X]$ and $[0, 0, X, 0]$.
$C[8]$ contains the compositions of the form $[0, 0, 0, X]$.

The variables used in this procedure are defined as follows:

C is the list described above.
N is a list of 4-compositions of n.
$NNum$ is the number of elements in N.

```
jan_composition4:=proc(n) local C,N,NNum;

    # C[1] contains the compositions of the form [X,X,X,X].
    C[1]:=[op(composition(n,4))];

    N:=composition(n,3);
    NNum := nops(N);
    # C[1] contains the compositions of the form [X,X,X,0].
    C[1] := [op(C[1]),seq([N[i][1],N[i][2],N[i][3],0],i=1..NNum)];

    # C[2] contains the compositions of the form [X,X,0,X].
    C[2] := [seq([N[i][1],N[i][2],0,N[i][3]],i=1..NNum)];

    # C[3] contains the compositions of the form [X,0,X,X].
    C[3] := [seq([N[i][1],0,N[i][2],N[i][3]],i=1..NNum)];
```

```
# C[4] contains the compositions of the form [0,X,X,X].
C[4] := [seq([0,N[i][1],N[i][2],N[i][3]],i=1..NNum)];

#Construct the compositions with 2 nonzero entries.
N:=composition(n,2);
NNum := nops(N);
# C[2] contains the compositions of the form [X,X,0,0].
C[2] := [op(C[2]),seq([N[i][1],N[i][2],0,0],i=1..NNum)];

# C[3] contains the compositions of the form [X,0,X,0].
C[3] := [op(C[3]),seq([N[i][1],0,N[i][2],0],i=1..NNum)];

# C[4] contains the compositions of the form [0,X,X,0].
C[4] := [op(C[4]),seq([0,N[i][1],N[i][2],0],i=1..NNum)];

# C[5] contains the compositions of the form [X,0,0,X].
C[5] := [seq([N[i][1],0,0,N[i][2]],i=1..NNum)];

# C[6] contains the compositions of the form [0,X,0,X].
C[6] := [seq([0,N[i][1],0,N[i][2]],i=1..NNum)];

# C[7] contains the compositions of the form [0,0,X,X].
C[7] := [seq([0,0,N[i][1],N[i][2]],i=1..NNum)];

#Construct the compositions with 1 nonzero entry.
# C[5] contains the compositions of the form [X,0,0,0].
C[5] := [op(C[5]),[n,0,0,0]];

# C[6] contains the compositions of the form [0,X,0,0].
C[6] := [op(C[6]),[0,n,0,0]];

# C[7] contains the compositions of the form [0,0,X,0].
C[7] := [op(C[7]),[0,0,n,0]];

# C[8] contains the compositions of the form [0,0,0,X].
C[8] := [[0,0,0,n]];

op(C);
end proc:
```

D.10. Counting Janiform Codes. Recall Definition A.9.

Algorithm D.15. *This procedure counts the number of (m,n) janiform codes up to isomorphism. By Theorem 4.20,*

$$\sum_{M,N} Bipartite\_Strict\,(M[2], N[2]) \cdot Bipartite\_Strict\,(M[3], N[3]),$$

where the sum is taken over all nonnegative compositions $M = (M[1], M[2], M[3], M[4])$ and $N = (N[1] = M[1], N[2], N[3], N[4])$ of m and n, respectively, such that $M[j]$ and $N[j]$ are either both

zero or both nonzero for each $j \in \{1,2,3\}$. We note that $E(i,j) = Bipartite\_Strict$ for $i,j = 0$ and for $i,j \in Z^+$.

The variables used in this procedure are defined as follows:

M and N contain lists of lists of compositions of m and n, respectively.
$MNum$ and $NNum$ contain the length of the lists $M[i]$ and $N[i]$, respectively.
$StartIndex$ is the smallest positive integer such that $M[StartIndex]$ and $N[StartIndex]$
    are nonempty lists.
i, j, and k are loop indices.
$NumCodes$ contains the number of codes counted so far.

```
janiform_code:=proc(m,n) local M, N, MNum, NNum, i, j, k, StartIndex, NumCodes;

   if m = 0 or n = 0 then
       return(0);
   end if;

   # M and N contain the nonnegative 4-compositions of m and n, respectively, such that
   # M[i] and N[i] contain a list of partitions such that M[i][j] and N[i][k] satisfy
   # M[i][j][q] and N[i][k][q] are both zero or both nonzero for each q in {1,2,3}.
   M := jan_composition4(m);
   N := jan_composition4(n);

   if m > 2 and n > 2 then
       #Each M[i] and N[i] list contains at least one element.
       StartIndex := 1;
   elif m = 1 or n = 1 then
       # We know that either M[1],M[2],M[3], and M[4] are empty or N[1], N[2], N[3], and N[4]
       # are empty. Thus, we need to start with index 5.
       StartIndex := 5;
   else # m = 2 or n = 2
       #We know that either M[1] or N[1] is empty. Thus, we need to start with index 2.
       StartIndex := 2;
   end if;

   NumCodes := 0;
   for i from StartIndex to 8 do
       MNum := nops(M[i]);
       NNum := nops(N[i]);
       for j from 1 to MNum do
           for k from 1 to NNum do
               if M[i][j][1] = N[i][k][1] then
                   NumCodes := NumCodes + Bipartite_Strict(M[i][j][2],N[i][k][2])
                       + Bipartite_Strict(M[i][j][3],N[i][k][3]);
               end if;
           end do;
       end do;
   end do;
   NumCodes;
end proc:
```

D.11. Counting Self-Companion Codes. Recall Definition A.5.

Algorithm D.16. *This procedure counts the number of (m, n) self-companion codes up to isomorphism. By Theorem 4.16, the number is $s(m, n)$.*

The variables used in this procedure are defined as follows:

NumCodes contains the number of precodes.

self_companion_code:=proc(m,n) local NumCodes;

```
   if m = 0 or n = 0 then
      return(1);
   end if;

   NumCodes := s(m,n);
end proc:
```

D.12. Counting Self-Opposite Codes. Recall Definition A.7.

Algorithm D.17. *This procedure computes all of the nonnegative 4-compositions of n. It returns them in a list C.*

The variables used in this procedure are defined as follows:

C is the list described above.
N is a list of 4-compositions of n.
$NNum$ is the number of elements in N.

composition4:=proc(n) local C,N,NNum;

```
   # C contains the compositions of the form [X,X,X,X].
   C :=[op(composition(n,4))];

   N:=composition(n,3);
   NNum := nops(N);
   # Add the compositions of the form [X,X,X,0].
   C := [op(C),seq([N[i][1],N[i][2],N[i][3],0],i=1..NNum)];

   # Add the compositions of the form [X,X,0,X].
   C := [op(C),seq([N[i][1],N[i][2],0,N[i][3]],i=1..NNum)];

   # Add the compositions of the form [X,0,X,X].
   C := [op(C),seq([N[i][1],0,N[i][2],N[i][3]],i=1..NNum)];

   # Add the compositions of the form [0,X,X,X].
   C := [op(C),seq([0,N[i][1],N[i][2],N[i][3]],i=1..NNum)];

   #Construct the compositions with 2 nonzero entries.
   N:=composition(n,2);
   NNum := nops(N);
   # Add the compositions of the form [X,X,0,0].
   C := [op(C),seq([N[i][1],N[i][2],0,0],i=1..NNum)];
```

```
# Add the compositions of the form [X,0,X,0].
C := [op(C),seq([N[i][1],0,N[i][2],0],i=1..NNum)];

# Add the compositions of the form [0,X,X,0].
C := [op(C),seq([0,N[i][1],N[i][2],0],i=1..NNum)];

# Add the compositions of the form [X,0,0,X].
C := [op(C),seq([N[i][1],0,0,N[i][2]],i=1..NNum)];

# Add the compositions of the form [0,X,0,X].
C := [op(C),seq([0,N[i][1],0,N[i][2]],i=1..NNum)];

# Add the compositions of the form [0,0,X,X].
C := [op(C),seq([0,0,N[i][1],N[i][2]],i=1..NNum)];

#Construct the compositions with 1 nonzero entry.
# Add the compositions [n,0,0,0],[0,n,0,0],[0,0,n,0], and [0,0,0,n].
C := [op(C),[n,0,0,0],[0,n,0,0],[0,0,n,0],[0,0,0,n]];
C;
end proc:
```

Algorithm D.18. *This procedure counts the number of (m,m) self-opposite codes up to isomorphism. By Theorem 4.21,*

$$\sum_M E\left(M[2], M[3]\right),$$

where the sum is taken over all nonnegative compositions $M = (M[1], M[2], M[3], M[4])$ of m such that $M[2]$ and $M[3]$ are either both zero or both nonzero. (Note that when $i, j = 0$, then $E(i,j) = 1$, and when $i, j > 0$, $E(i,j) = Bipartite\_Strict(i,j)$).

The variables used in this procedure are defined as follows:

M contains lists of lists of compositions of m.
$MNum$ contains the length of the lists $M[i]$.
i is a loop index.
$NumCodes$ contains the number of codes counted so far.

```
self_opposite_code:=proc(m) local M, MNum, i, NumCodes;

    if m = 0 then
        return(1);
    end if;

    # M contains the nonnegative 4-compositions of m.
    M := composition4(m);
    MNum := nops(M);

    NumCodes := 0;
    for i from 1 to MNum do
        # Recall that we must ignore the composition M[i] if one of M[i][2] and M[i][3] is zero
        # and the other is nonzero. Recall also that E(i,j)=Bipartite_Strict(i,j) if i,j > 0,
```

```
      # and E(i,j)=1 if i,j = 0.
      if (M[i][2] > 0) and (M[i][3]) > 0 then
         NumCodes := NumCodes
             + Bipartite_Strict(M[i][2],M[i][3]);
      elif (M[i][2] = 0) and (M[i][3]) = 0 then
         NumCodes := NumCodes + 1;
      end if;
   end do;
   NumCodes;
end proc:
```

D.13. Counting All Precodes.

We now count all precodes up to isomorphism.

Algorithm D.19. *This procedure counts the number of (m, n) precodes up to isomorphism. By Lemma 5.1, the number is $mixed\_bipartite(m, n)$.*

The variables used in this procedure are defined as follows:

NumPrecodes contains the number of precodes.

```
precode:=proc(m,n) local NumPrecodes;

   if m = 0 or n = 0 then
      return(1);
   end if;

   NumPrecodes := mixed_bipartite(m,n);
end proc:
```

D.14. Counting Self-Companion Precodes.

Recall Definition A.5.

Algorithm D.20. *This procedure counts the number of (m, n) self-companion precodes up to isomorphism. By Theorem 5.2, the number is $bipartite(m, n)$.*

The variables used in this procedure are defined as follows:

NumPrecodes contains the number of precodes.

```
self_companion_precode:=proc(m,n) local NumPrecodes;

   if m = 0 or n = 0 then
      return(1);
   end if;
   NumPrecodes := bipartite(m,n,[m]);
end proc:
```

D.15. Counting Janiform Precodes.

Recall Definition A.9.

Algorithm D.21. *This procedure counts the number of (m, n) janiform precodes up to isomorphism. By Theorem 5.3, the number is $code(m, n)$.*

The variables used in this procedure are defined as follows:

NumPrecodes contains the number of precodes.

janiform_precode:=proc(m,n) local NumPrecodes;

    NumPrecodes := code(m,n,false);
end proc:

D.16. Counting Self-Opposite Precodes. Recall Definition A.7.

Algorithm D.22. *This procedure counts the number of (m, m) self-opposite precodes up to isomorphism. By Theorem 5.4, the number is bipartite(m, m).*

The variables used in this procedure are defined as follows:

NumPrecodes contains the number of precodes.

self_opposite_precode:=proc(m) local NumPrecodes;

    if m = 0 then
       return(1);
    end if;
    NumPrecodes := bipartite(m,m,[m]);
end proc:

VITA

TRAE DOUGLAS HOLCOMB

EDUCATION

Ph.D. in Mathematics, Texas A&M University, May 2002.
M.S. in Applied Mathematics, University of Colorado at Colorado Springs, August 1997.
B.S. in Computer Science, Southwest Texas State University, December 1991.

PUBLICATIONS

G. R. Blakley, I. Borosh, T. Holcomb, and A. Klappenecker, Categorical code constructions, in preparation.

J. Haefner and T. Holcomb, The picard group of a structural matrix algebra, *Linear Algebra and Its Applications* **304** (2000), 69-101.